



Katedra: *Algorytmów i Modelowania Systemów*

Imię i nazwisko dyplomanta: *Tomasz Goluch*

Nr albumu: *103057*

Forma i poziom studiów: *Dzienne, magisterskie*

Kierunek studiów: *Informatyka*

Praca dyplomowa magisterska

Temat pracy: *Metody komputerowego rozwiązywania gier logicznych*

Temat (w języku angielskim): *Computer methods of solving logical games*

Kierujący pracą: *dr hab. inż. Krzysztof Giaro, prof. nadzw. PG*

Zakres pracy:

Przegląd znanych algorytmów określania wartości pozycji w niesosowych grach dwóch graczy z pełną informacją. Wybór gry i zaprogramowanie generatora ruchów. Zaprogramowanie algorytmu określającego wartość gry. Przeprowadzenie eksperymentu obliczeniowego.

Dziękuję panu dr hab. inż. Krzysztofowi Giaro za nieoceniony wkład w tę pracę. Za to, że wytrwale i konsekwentnie rozświetlał mi umysł widząc mnie błądzącego po omacku w nieznanym mi dotąd labiryncie.

Dziękuję panu dr inż. Robertowi Janczewskiemu za cierpliwość podczas recenzji jak i również za rzetelną opinię oraz konstruktywne uwagi.

Dziękuję wszystkim osobom które pomagały mi podczas pisania tej pracy, w szczególności: Milenie Stępień, Dagmarze Weynie, Bożeniu Pruskiej, Magdalenie Loranty, Danieli Roman, Arkadiuszowi Roman, Stanisławowi Ciepielewskiemu, Kubie Łopatce oraz wszystkim innym za motywację i demotywacje :)

dedykuję rodzicom

„Not everything that counts can be counted,
and not everything that can be counted counts.”

– Albert Einstein

Oświadczenie

Oświadczam, że niniejszą pracę dyplomową wykonałem samodzielnie. Wszystkie informacje umieszczone w pracy uzyskane ze źródeł pisanych oraz informacje ustne pochodzące od innych osób zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami.

.....
podpis dyplomanta

Spis treści

Oświadczenie	vii
Streszczenie	xxi
Abstract	xxiii
1. Wprowadzenie	1
2. Historia rozwiązywania gier logicznych	5
2.1. Podział gier	5
2.2. Rozwiązać grę – co to znaczy?	11
2.3. Gry rozwiążane	13
2.4. Perspektywy osiągnięć w rozwiązywaniu gier	36
3. Metody rozwiązywania gier logicznych	43
3.1. Złożoność przestrzeni stanów i drzewa gry	44
3.2. Analiza wsteczna	53
3.3. Algorytmy <i>depth-first search</i>	57
3.4. Struktury reprezentujące wiedzę o grze	64
3.5. Algorytmy <i>best-first search</i>	70
3.6. Inne wersje algorytmu „Proof-number search”	108
4. Gra Skoczki	111
4.1. Dlaczego <i>Skoczki</i> ?	111
4.2. Oryginalne zasady gry	113
4.3. Poprawione zasady gry	114
4.4. Inne warianty gry	116
5. Program Weles	119
5.1. Środowisko programistyczne	119
5.2. Architektura programu	119
5.3. Interfejs użytkownika	121
5.4. Moduł zarządzania	125
5.5. Generator ruchów – moduł silnika gry	127

5.6. Pakiet algorytmów rozwiązywających – moduł sztucznej inteligencji	129
5.7. Aspekty dynamiczne programu	130
6. Wyniki eksperymentu obliczeniowego	137
6.1. Własności gry <i>Skoczki</i>	137
7. Podsumowanie	149
A. Przestrzeń stanów gry <i>Skoczki</i>	151
B. Zasady omawianych gier	155
B.1. Aukcja o dolara	155
B.2. Awari	155
B.3. Czwórki	156
B.4. Dakon-6	156
B.5. Dylemat więźnia	158
B.6. Fanorona	158
B.7. Go-Moku	159
B.8. Go-Moku <i>wolne</i>	160
B.9. Gra w życie	160
B.10. Halma	160
B.11. Hex	161
B.12. Hex konkurencyjny	162
B.13. Kalah(6,4)	162
B.14. Latrunculi	163
B.15. Maharadża i Sepoje	163
B.16. Młynek	164
B.17. Nim	165
B.18. Orzeł i reszka	165
B.19. Pentomino	166
B.20. Quartostandardowe	166
B.21. Quarto <i>twist</i>	166
B.22. Qubic	166
B.23. Renju	167
B.24. Teeko	168
B.25. Tygrysy i Kozy	169
B.26. Warcaby <i>angielskie</i>	170
B.27. Wilk i owce	171

C. Zawartość płyty DVD-ROM **173**

Bibliografia **175**

Spis tabelic

2.1.	Hipotezy z 1990 na 2000 rok, dotyczące siły programów, grających w gry z olimpiad komputerowych	19
2.2.	Wyliczone wartości optymalnej rozgrywki gry Kalach(m,n)	27
2.3.	Pojedynek w <i>Renju</i> , gracze używają optymalnych strategii	27
2.4.	Lista rozwiązanych gier logicznych dwuosobowych z pełną informacją	35
2.5.	Hipotezy z 2000 na 2010 rok dotyczące siły programów grających w gry z olimpiad komputerowych	36
3.1.	Podział na podprzestrzenie stanów gry <i>Kółko i krzyżyk</i>	47
3.2.	Podział na podprzestrzenie stanów gry <i>Młynek</i>	48
3.3.	Złożoność podprzestrzeni stanów gry <i>Warcaby angielskie</i> [50]	49
5.1.	Sześć wariantów gry <i>Skoczki</i>	128
6.1.	Górne oszacowanie rozmiaru przestrzeni stanów gry <i>Skoczki</i> o różnych wymiarach	139
6.2.	Porównanie oszacowania rozmiaru przestrzeni stanów gry <i>Skoczki</i> z rzeczywistymi wartościami	140
6.3.	Porównanie oszacowania rozmiaru przestrzeni stanów gry <i>Skoczki</i> z rzeczywistymi wartościami (wartości względne)	141
6.4.	Wartość pozycji startowej gry <i>Skoczki</i> w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.1	142
6.5.	Wartość pozycji startowej gry <i>Skoczki</i> w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.2	143
6.6.	Wartość pozycji startowej gry <i>Skoczki</i> w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.3	145

Spis rysunków

2.1.	Plansza do gry w <i>Hex'a</i> (11×11)	13
2.2.	Plansza do <i>Hex'a</i> , przedstawiona w formie grafu	14
2.3.	Zapełniona plansza do <i>Hex'a</i> , przedstawiona w formie grafu	14
2.4.	Przykład nie zakończonej rozgrywki w <i>Hex'a</i> z jedynym wolnym polem	17
2.5.	Różne wcielenia tej samej gry <i>Qubic</i>	18
2.6.	Gra <i>Czwórki</i>	19
2.7.	Przykładowa rozgrywka w grze <i>Go-Moku</i>	20
2.8.	Gra <i>Młynek</i>	22
2.9.	Pionki do gry w <i>Pentomino</i>	22
2.10.	Startowe pozycje w <i>Pentomino</i> zapewniające wygraną rozpoczynającemu	23
2.11.	Plansza i pionki do gry <i>Quarto</i>	24
2.12.	Plansza do gry <i>Teeko</i>	24
2.13.	Plansza do gry z rodziny <i>Mankala</i>	25
2.14.	Plansza do gry <i>Ohvalhu</i> (Dakon-8 prawoskrętny)	25
2.15.	Dziesięć pierwszych stanów gry otwarcia wygrywającego w <i>Ohvalhu</i> (cyfra na danym polu oznacza liczbę znajdujących się w nim ziaren) .	26
2.16.	Gra <i>Renju</i>	28
2.17.	Plansza do gier <i>Mankala</i> : <i>Awari</i> , <i>Dakon-6</i> i <i>Kalah(6,4)</i>	29
2.18.	Pozycje w <i>Warcabach angielskich</i>	30
2.19.	Przestrzeń stanów w <i>Warcabach angielskich</i> [51]	31
2.20.	Pozycja początkowa w <i>Fanorонie</i>	32
2.21.	Średni współczynnik rozgałęzienia jako funkcja liczby pionków na planszy [49] występujący w grze <i>Farona</i>	33
2.22.	Gra <i>Tygrysy i Kozy</i>	33
2.23.	Ruch tygrysów, jednak kozy wygrywają w 5 posunięciach	34
2.24.	Poster ze strony domowej Joela Feinstein'a	36
2.25.	Pozycja początkowa najdłuższej dotychczas znalezionej końcówki w <i>Warcabach polskich</i> [20]	37
2.26.	Przykładowa końcówka w <i>Szachach</i>	38
2.27.	Pozycja początkowa najdłuższej rozgrywki w <i>Szachach</i> (3×3) [31] .	38

2.28. Pozycja początkowa w grze <i>Maharadža i Sepoje</i> [10]	39
2.29. Rozwiązane ruchy otwierające w <i>Hex'ie</i> o rozmiarach do 6×6 pól włącznie (kolor przedstawia zwycięzcę, jeśli czerwony rozpoczyna na danym polu)	40
2.30. Rozwiązane ruchy otwierające w <i>Hex'ie</i> (7×7) (kolor przedstawia zwycięzcę, jeśli czerwony rozpoczyna na danym polu)	40
2.31. Rozwiązane ruchy otwierające w <i>Hex'ie</i> (8×8) (kolor przedstawia zwycięzcę, jeśli czerwony rozpoczyna na danym polu)	41
2.32. Faza końcowa gry <i>Go</i>	41
2.33. Trzy przykłady optymalnej rozgrywki w <i>Go</i> (5×5) [62]	42
3.1. Plansza do <i>Halmy</i> przedstawiająca pozycję początkową dla dwóch graczy	46
3.2. Podział gry <i>Młynek</i> [19]	47
3.3. Przykłady nieosiągalnej pozycji w grze <i>Warcaby</i>	50
3.4. Przykłady symetrycznych stanów gry <i>Kółko i krzyżyk</i>	51
3.5. Plansza do gry <i>Młynek</i>	51
3.6. Szesnaście symetrycznych stanów w grze <i>Młynek</i>	52
3.7. Drzewo minimaksowe	59
3.8. Przycięte drzewo minimaksowe	60
3.9. Przycięte drzewo minimaksowe	60
3.10. Drzewo negamaksowe	62
3.11. Drzewo negamaksowe z odcinaniem $\alpha-\beta$	63
3.12. Drzewo AND/OR	67
3.13. Digraf acykliczny AND/OR	68
3.14. Digraf cykliczny AND/OR	69
3.15. Drzewo AND/OR z liczbami dowodzącymi	72
3.16. Drzewo AND/OR z liczbami obalającymi	74
3.17. Drzewo AND/OR z najbardziej dowodzącym wierzchołkiem	75
3.18. Przykład transpozycji	85
3.19. Dwukrotnie policzony wierzchołek obalający	86
3.20. Zignorowany lepszy wierzchołek obalający	87
3.21. Konwersja DCG do drzewa	90
3.22. Problem nieskończonej aktualizacji w DCG	91
3.23. Problem opóźnionego wyliczenia wartości wierzchołka w DCG	92
3.24. Przeoczona wygrana	94
3.25. Kroki 1-3 wykonywania funkcji <i>SelectMostProving(node)</i>	98

3.26. Kroki 4-6 wykonywania funkcji <i>SelectMostProving(node)</i>	99
3.27. Kroki 7-8 wykonywania funkcji <i>SelectMostProving(node)</i>	100
4.1. Plansza do gry <i>Latrunculi</i> zawierająca pozycję startową	112
4.2. Plansza gry <i>Skoczki</i> (wg L.Pijanowskiego) [40]	114
4.3. Przykładowe posunięcie wg trywialnej strategii remisowej	115
4.4. Plansza do gry <i>Skoczki</i>	117
5.1. Diagram komponentów programu Weles	120
5.2. Diagram klas interfejsu użytkownika	122
5.3. Opis interfejsu użytkownika – trwająca rozgrywka	123
5.4. Opis interfejsu użytkownika – wygrana białych	124
5.5. Opis interfejsu użytkownika – rozwiązywanie pozycji startowej	125
5.6. Diagram klas modułu zarządzania	126
5.7. Diagram klas modułu silnika gry	127
5.8. Powiązanie pól planszy z bitami typu <i>ulong</i>	128
5.9. Diagram klas modułu algorytmów rozwiązywających	130
5.10. Diagram Sekwencji – Inicjalizacja GUI	131
5.11. Diagram Sekwencji – start rozgrywki	132
5.12. Diagram Sekwencji – ruch człowieka (białymi)	133
5.13. Diagram Sekwencji – ruch komputera	135
6.1. Nieosiągalna pozycja w każdym wariantie gry <i>Skoczki</i> pozwalającym jedynie na skoki	138
6.2. Wykres funkcji współczynnika regulującego liczbę wierzchołków drzewa poziomu drugiego	144
6.3. Drzewo gry dla planszy o wymiarach (1×7)	146
6.4. Digraf dowodzący remis dla planszy o wymiarach 1×8	148
A.1. Pełna przestrzeń stanów gry <i>Skoczki</i> (1×8) (górska połowa)	152
A.2. Pełna przestrzeń stanów gry <i>Skoczki</i> (1×8) (dolna połowa)	153

Spis algorytmów

1.	Analizy wstępnej	54
2.	Generujący wszystkie możliwe stany gry	55
3.	Kroku analizy wstępnej	56
4.	Oznaczający nieznane stany jako remisowe	57
5.	Minimax	58
6.	Negamax	61
7.	Minimax z przycinaniem $\alpha-\beta$	64
8.	Negamax z przycinaniem $\alpha-\beta$	65
9.	Proof-Number Search	76
10.	Ustalający liczby proof i disproof	77
11.	Wybierający najbardziej dowodzący wierzchołek	79
12.	Rozwijający wierzchołek (<i>wersja z natychmiastowym wyliczaniem</i>) . .	79
13.	Rozwijający wierzchołek (<i>wersja z opóźnionym wyliczaniem</i>)	80
14.	Uaktualniający przodków	80
15.	Rozwijający wierzchołek w algorytmie pn^2 -search	82
16.	Proof-Number Search z wierzchołkiem bieżącym	84
17.	Uaktualniający przodków (rozszerzony)	84
18.	Rozwijający wierzchołek z uwzględnieniem transpozycji (<i>wersja z natychmiastowym wyliczaniem</i>)	88
19.	Rozwijający wierzchołek z uwzględnieniem transpozycji i powtórzeń (<i>wersja z natychmiastowym wyliczaniem</i>)	95
20.	BTA Proof-Number Search	97
21.	BTA wybierający najbardziej dowodzący wierzchołek	101
22.	BTA wybierający najbardziej dowodzący wierzchołek cd.	103
23.	BTA wybierający najlepszy wierzchołek potomny	105
24.	BTA rozwijający wierzchołek	106
25.	BTA uaktualniający przodków	107
26.	BTA uaktualniający wierzchołki OR	108
27.	BTA uaktualniający wierzchołki AND	109

Streszczenie

W 2007 roku Jonathan Schaeffer odniósł spektakularny sukces rozwiązując, popularny w Angli i USA, wariant Warcabów zwany *English draughts* lub *American checkers*. Jednak już wcześniej rozwiązano wiele znanych gier planszowych, takich jak: *Młynek*, *Czwórki*, *Pentomino*, *Avari*, *Go-Moku* czy *Renju*. Niniejsza praca ma na celu przybliżyć czytelnikowi co kryje się pod terminem „rozwiązać grę”, czy to oznacza, że wiemy już wszystko na jej temat, że odgadliśmy już wszystkie jej tajemnice?

Praca jest podzielona na cztery części. Na początku pokazuje jak coraz bardziej zaawansowane metody algorytmiczne oraz postępująca siła obliczeniowa komputerów przyczyniła się do spektakularnych sukcesów w dziedzinie rozwiązywania gier. Jest to swego rodzaju historia zmagań ludzi z bardzo złożonymi problemami obliczeniowymi.

W drugiej części przedstawia ogólne metody algorytmiczne rozwiązywania gier. Jest to droga wytyczona już w 1950 roku przez Claude E. Shannon'a w jego fundamentalnym dziele „Programming a Computer for Playing Chess”. Od tamtych czasów powstało wiele udoskonaleń i nowych pomysłów, jednak zachwycające jest to, że ogólne przesłanie Shannon'a po dzisiejszy dzień jest jak najbardziej aktualne.

Trzecia część pracy jest próbą praktycznego zmierzenia się z problematyką rozwiązywania gier. Polegała na zaprogramowaniu i rozwiązaniu prostej gry logicznej. W tej części autor przedstawia strukturę blokową własnego programu, zapoznaje czytelnika z zastosowanym środowiskiem programistycznym, ciekawszymi szczegółami implementacyjnymi oraz zastosowanymi metodami algorytmicznymi.

Na końcu przedstawiono wyniki przeprowadzonego eksperymentu obliczeniowego dla wybranej gry.

Abstract

In 2007 Jonathan Schaeffer spectacularly succeeded in solving a famous variation of Draughts in USA and England called *English draughts* or *American checkers*. However, numerous well-known boarding games were already solved such as: *Nine Men's Morris*, *Connect-Four*, *Pentominoes*, *Avari Go-Moku* or *Renju*. This thesis aims to focus on what the term „solving a game” means. Does it signify that we know all about its issue, and then have we guessed all its mysteries?

The thesis is divided into four parts. The first part shows how far the increasingly advanced algorithmic methods and progressive computer’s computing power has gone towards reaching the spectacular success in the domain of computing games. It is a storyline of people’s struggle with very complex computational problems.

The second part presents general algorithmic methods of solving games. This is a path which has already been marked out in 1950 by Claude E. Shannon in his fundamental work ”Programming a Computer for Playing Chess”. From those years on there have been plenty of improvement and new ideas. Yet the most amazing fact is that the general Shannon’s message is up-to-date until today’s times.

The third part is a practical attempt at dealing with problems of solving games. It is planned to demonstrate a single programming and solving of a simple logical game. In this section author sets to illustrate the block structure of his own program, acquaints the reader with the used program environment, interesting implementation details and the adopted algorithmic methods.

Finally, the results of carried out calculations experiment for a specifically chosen game are presented.

Rozdział 1

Wprowadzenie

Ludzie często posługują się słowami, których znaczenie wydaje się oczywiste i nie wywołuje problemów z ich zrozumieniem u odbiorcy. Jednak już definicja równościowa (klasyczna) takiego słowa stanowi niemały problem. Jednym z przykładów jest słowo *gra*.

Najprawdopodobniej pierwszym z filozofów uniwersyteckich, który próbował odpowiedzieć na pytanie: „Czym jest gra?” jest Ludwig Wittgenstein. W swoim dziele „Dociekania filozoficzne”[66] określa on termin „gra” jako trudny do zdefiniowania przy pomocy pojedynczej definicji. Twierdzi, że na gry trzeba patrzeć jak na swojego rodzaju rodzinę, w której „widzimy skomplikowaną siatkę zachodzących na siebie i krzyżujących się podobieństw; podobieństw w skali dużej i małej”.

Intuicyjnie określenie gra kojarzy nam się z grą towarzyską. Wiemy, że gry nieroziłącznie towarzyszyły ludzkości. Najstarsze odkryte rekwizyty służące do grania datowane są na 2600 rok p.n.e. Gry towarzyskie spełniały różne funkcje w życiu człowieka: pozwalały się odpreżyć, rozwinać intelektualnie, ale również stoczyć bezkrwawy pojedynek bądź wygrać niewolnika. Pewne z nich przechodziły w zapomnienie, inne ewoluowały dając miejsce nowym, a wiele z nich zachowało się do dzisiaj. Fascynującym jest fakt, że nawet przy prostych zasadach, strategia gracza może być doskonala przez całe życie a nawet przez pokolenia, gdzie wiedza przekazywana jest z mistrza na ucznia.

Jednak spektrum słowa gra jest dużo szersze. Przykładowo, konflikty militarne którymi przesiąknięta jest historia ludzkości, są typowym zjawiskiem dającym się opisać na przykładzie gry. Permanentne doskonalenie tej strategii doprowadziło do paradoksu znanego pod nazwą (MAD)¹, polegającego na tym, że optymalna strategia wymaga jedynie posiadania środków wywołujących największe spustoszenie w zasobach przeciwnika. Żaden z graczy nie przewiduje ich użycia jako pierwszy, ponieważ wpływa to bardzo negatywnie na wartość wypłaty każdego z graczy. Zatem obydwie (używając identycznej strategii) uzyskują równowagę teoretycznie gwarantującą pokój.

¹Akronim od *Mutual Assured Destruction* – Wzajemne Zagwarantowane Zniszczenie, ale również *mad* – szalony. Strategia polegająca na odstraszaniu przeciwnika, wizją jego unicestwienia w przypadku użycia przez niego broni nuklearnej.

Jednakże, życie na ziemi od początku swojego istnienia opiera się na walce, w której tylko gatunki potrafiące przystosować się do istniejących warunków mogły przetrwać. Innym przykładem gry o przetrwanie jest fakt czystego altruizmu wśród zwierząt który sumarycznie maksymalizuje zysk populacji. Zjawisko to jest widoczne podczas dzielenia się przez nietoperze krwią z polowania. Pozwala ono przetrwać osobnikom, którym nie powiodło się podczas łowów.

Richard Dawkins [14] w książce „Samolubny gen” stawia kontrowersyjną tezę oznajmującą, że to geny są głównymi graczami, a organizmy w których się one znajdują są jedynie ich maszynami przetrwania, pozwoliło to na wyjaśnienie zjawiska zwanego altruizmem krewnowym. Jego przykłady można zaobserwować wśród różnych gatunków owadów, jednym z przykładów jest poświęcanie się robotnic dla swojej królowej, która jest jedynym źródłem przetrwania genów tego gatunku. W podobny sposób próbuje się tłumaczyć zachowania wśród ludzi, takie jak poświęcenie dla dziecka bądź członków najbliższej rodziny, czy liczne akty heroizmu podczas wojen.

Cofając się jeszcze bardziej w czasie możemy natrafić na jedną z pierwszych gier rozgrywającą się w naszym kosmosie. Jej początek jest datowany na 300 tysiąclecie istnienia wszechświata, czyli początek ery galaktycznej, kiedy to bardzo równomiernie, lecz nie idealnie, rozłożona materia we wszechświecie stała się celem dwóch przeciwnych graczy: siły grawitacji przyciągającej ją do siebie i ekspansji wszechświata powodującej oddalanie się tejże materii. Wynikiem tej gry jest dzisiejszy świat który, możemy obserwować pod postacią olbrzymiej liczby galaktyk. Ta gra toczy się po dzisiejsze czasy a jej wynik nadal jest nieznany. Materia o której mowa, jest jedynie pozostałością po jej anihilacji z antymaterią. Innymi słowy po grze, która rozegrała się pomiędzy tymi dwoma graczami w 10^{-35} części sekundy istnienia wszechświata. Jej wynik tłumaczymy zjawiskiem łamania symetrii.

Znamion gry możemy doszukiwać się jeszcze wcześniej, idąc śladami amerykańskiego fizyka Lee Smolin'a [58]. Uważa on, że podobnie, jak to ma miejsce w przypadku ewolucji biologicznej, dobór naturalny realizowany jest również w skali kosmologicznej, a zapadnięcie się czarnej dziury jest początkiem nowego wszechświata w którym mutują stałe fizyczne, tym samym podlegając selekcji naturalnej. Te wszechświaty, w których będą korzystniejsze warunki do produkcji kolejnych czarnych dziur, przetrwają poprzez ich masową produkcję.

Wracając do bliższych nam czasów, również zachowanie się wolnego rynku można obrazować przy pomocy gry. Szczególną rolę odgrywa tutaj nowa gałąź matematyki zwana „teorią gier”. Definiuje ona grę jako sytuację konfliktową i bada jakie strategie powinni przyjąć gracze aby osiągnąć najlepsze wyniki. Według

Guillermo Owena [38] gra powinna mieć swój punkt początkowy, po którym następuje ciąg kolejnych posunięć, w każdym z nich jeden z graczy dokonuje wyboru z kilku możliwości a dodatkowo dopuszczane są jeszcze posunięcia losowe. Przy końcu gry powinna wystąpić wypłata dla każdego z graczy. Przeważnie w takiej grze czyjaś wygrana oznacza czyjąś przegrana. Istnieją jednak gry pozwalające na tworzenie koalicji, w których współpraca graczy pozwala na osiągnięcie lepszego wyniku poprzez każdego z nich, niż rezultaty które mogliby osiągnąć podczas rywalizacji. Nawiąsem mówiąc, znajomość prowadzenia takiej taktyki byłaby szczególnie przydatna dla działaczy polskiej sceny politycznej.

Czasami ludzie uczestniczą w grach, w których na wynik rozgrywki większy wpływ ma szczęście niż umiejętności gracza. Ponadto, bardzo często prawdopodobieństwo wygranej w nich jest znikome. W takim przypadku również mamy do czynienia z rodzajem gry zwanym *hazardem* i to notabene gry bardzo niebezpiecznej dla uczestnika, szczególnie w długim horyzoncie czasowym. W sytuacji gdzie los ma wpływ na wynik, albo tam gdzie nie znamy sytuacji przeciwnika możemy jedynie domyślać się stanu rzeczywistej sytuacji oraz możliwości jej rozwoju. A zatem możemy jedynie przewidywać z pewnym prawdopodobieństwem możliwy wynik.

Jedyną grupą gier, w których wynik teoretycznie da się przewidzieć to takie, w których żaden element losowy nie ma wpływu na wynik końcowy, a każdy z graczy dysponuje taką informacją o pozostałych graczech, która jest istotna do obrania przez niego optymalnej strategii gry. W przypadku takich gier wyniki graczy wykonujących w każdym swoim posunięciu najlepszy możliwy ruch są z góry możliwe do przewidzenia. Łatwo jest to zauważyc na przykładzie gry *Kółko i krzyzyk*, która należy do wspomnianej rodziny. Uważny gracz szybko znajdzie sobie strategię zapewniającą mu przynajmniej remis. Jednak w przypadku *Szachów* nie jest to już takie oczywiste. Metodami znajdowania takich właśnie rozwiązań będę się zajmował w niniejszej pracy.

Rozdział 2

Historia rozwiązywania gier logicznych

2.1. Podział gier

Liczba gier jest ogromna. Tylko tych popularnych jest mnóstwo a każda z osobna, poprzez zmianę zasad, może posiadać nieskończoną liczbę wariantów. Dodatkowo gry bardzo się od siebie różnią. Skrajnym przykładem może być porównanie *Aukcji o dolara*¹ obrazującej ekonomiczny paradoks z *Pilką nożną*². Aby zapanować nad tym chaosem, wprowadzono wiele kryteriów podziału gier. Pozwalają one na zdefiniowanie pewnego zbioru cech dla gier którymi będziemy się zajmować. Najważniejsze z nich to:

- **Rodzaj gry** – planszowe, karciane, strategiczne, logiczne, ekonomiczne, matematyczne, hazardowe, sportowe... i wiele, wiele innych.
- **Liczba graczy** – istnieją gry dla ścisłe określonej liczby osób np:
bezosobowe – *Gra w życie*³;
jednoosobowe – *Samotnik*, *Kostka rubika*;
dwouosobowe – *Kółko i krzyżyk*, *Orzeł i reszka*⁴, *Nim*⁵, *Warcaby*⁶, *Szachy*,

¹Zasady *Aukcji o dolara* przedstawiono w dodatku B.1.

²Ze względu na ogromną popularność *Pilki nożnej* oraz nadzwyczaj wysoką znajomość przez społeczeństwo obowiązujących w niej zasad, autor zrezygnował z ich umieszczania. Jednakże pragnie zauważyć, że ta z pozoru prosta gra sportowa przeszła głęboką metamorfozę. Stając się złożoną, wieloetapową grą o charakterze społeczno-ekonomicznym, poszerzoną o wiele nowych, niesymetrycznych zasad. Pozwalającą na tworzenie, niekoniecznie jawnych koalicji, z dużą ilością ukrytej informacji i bardzo rozbudowaną funkcją wypłaty: piłkarzy, sędziów, prezesów i innych uczestników (czyt. graczy). Jak widać, można by się tutaj jakiegoś wspólnego akcentu z *Aukcją o dolara* doszukać, jednak nie występuje on w klasycznej wersji tej gry.

³Zasady *Gry w życie* przedstawiono w dodatku B.9.

⁴Szczegółowe zasady gry *Orzeł i reszka* opisano w zamieszczono B.18.

⁵Szczegółowe zasady gry *Nim* opisano w zamieszczono B.17.

⁶Ze względu na ograniczenia objętościowe tej pracy zrezygnowano z umieszczenia zasad popularnych gier zakładając, że czytelnik jest z nimi zaznajomiony bądź może skorzystać z szeroko dostępnej literatury na ten temat.

*Go, Brydż, Dylemat więźnia*¹.

trzyosobowe – *Trylma, 3-5-8.*

czteroosobowe – *Kierki.*

Istnieją też gry n -osobowe dla ścisłe określonej liczby osób. Może to być zarówno przetarg, w którym uczestniczy n interesantów jak i turniej towarzyskiej gry dwuosobowej, w którym rywalizuje n -graczy. Gry n -osobowe można również uzyskiwać modyfikując istniejące już zasady innych gier.

Ciekawostką są gry, w których biorą udział cztery osoby, takie jak: *Brydż, Kanasta, Focus dla czterech osób, Szachy czteroosobowe*. Z punktu widzenia teorii gier są to gry dwuosobowe, ponieważ interesy każdej z par wspólników są identyczne.

Liczba graczy może być określona jakimś przedziałem liczbowym np:

Poker – od dwóch do dziesięciu graczy;

Monopoly – od dwóch do sześciu graczy;

Domino – minimum dwie osoby.

W skrajnym przypadku liczba graczy może być nieskończona. Przykładem zastosowania takich gier może być modelowanie sytuacji wolnorynkowych w analizie ekonomicznej [38].

- **Suma wypłat** – Suma wypłat może być niezerowa lub zerowa.

- *Gra o niezerowej sumie wypłat*² – to gra w której zysk (strata) jednego z graczy nie musi oznaczać wspólnej straty(zysku) drugiego z graczy. Przykładem takiej gry jest *Dylemat więźnia, Walka płci* lub gry jednoosobowe, gdzie strata (zysk) jedynego gracza nie może być zrekompensowana z powodu braku rywala. W tego typu grach zasadnym jest prowadzenie negocjacji pomiędzy graczami oraz zawazywanie koalicji.
- *Gra o sumie zerowej*³ – jest szczególnym przypadkiem gry o sumie stałej. Cechuje się tym, że zysk jednego z graczy jest stratą innego. Do tej grupy zalicza się większość gier towarzyskich np. *Szachy, Go, Poker, Brydż, Pilka nożna*. W przeciwieństwie do gier o niezerowej sumie wypłat nie ma tutaj miejsca na negocjacje.

¹Szczegółowe zasady gry *Dylemat więźnia* opisano w zamieszczono B.5.

²Inaczej zwana grą częściowo konkurencyjną.

³Jest to szczególny przypadek szerszej klasy tzw. *gier ścisłe konkurencyjnych*.

- **Symetria** – według teorii gier gry symetryczne to takie, w których gracze są w tej samej sytuacji początkowej i mogą realizować te same strategie¹. W przypadku gier o sumie zerowej ich macierz wypłat może być przedstawiona za pomocą macierzy antysymetrycznej². Przykładem takiej gry jest *Dylemat więźnia*, tutaj każdy z graczy może zachować się identycznie, a strategią ściśle dominującą³ każdego z nich jest współpraca z wymarem sprawiedliwości. Większość dwuosobowych gier towarzyskich staje się symetryczna, jeśli o wyborze rozpoczynającego zadecydujemy jakimś sprawiedliwym czynnikiem losowym, może to być rzut monetą symetryczną. Oczywiście wszystkie gry, które nie są symetryczne są niesymetryczne.
- **Symetria zasad gry** – dla logicznych gier towarzyskich, którymi będziemy się w tej pracy szczegółowo zajmować, symetria ta jest spełniona, jeżeli jedyną różnicą zasad dla każdego z graczy jest moment wykonania przez nich pierwszego posunięcia. *Kółko i krzyzyk*, *Młynek*, *Szachy*, *Go* to gry spełniające tę symetrię, w przeciwieństwie do takich jak: *Wilk i owce*, *Maharadża i Sepoje*, *Renju* oraz *Tygrysy i kozy*.
- **Stronniczość** – gry bezstronne (z ang. *impartial*), przykładowo: *Nim*, *Quarto*, to gry, w których możliwe posunięcia w danym stanie gry zależą wyłącznie od aktualnej pozycji a nie zależą od gracza. W grach stronniczych (z ang. *partisan*), do których zalicza się większość gier towarzyskich pula możliwych posunięć zależy od gracza. Stronniczość w *Szachach* objawia się tym, że grający białymi wykonuje ruchy jedynie białymi bierkami a grający czarnymi jedynie bierkami czarnymi.
- **Informacja** – informacja do jakiej ma dostęp gracz może się znacznie różnić w zależności od gry.
 - *Informacją kompletną* – nazywamy znajomość przez graczy swojej funkcji wypłat⁴, możliwych strategii wszystkich graczy oraz swojego stanu na każdym etapie gry [41].

¹Strategia, to najprościej mówiąc, założone z góry zachowanie się gracza, w każdym z osiągalnych stanów gry.

²Macierz antysymetryczna, zwana inaczej skośnie symetryczną, to macierz kwadratowa A spełniająca warunek: $A^T = -A$.

³Strategia ściśle dominująca gracza, to taka która bez względu na posunięcie przeciwnika jest dla gracza najkorzystniejsza.

⁴Funkcja wypłaty, to funkcja przypisująca pewną liczbę (wypłatę) każdemu z osiągalnych, końcowych stanów gry.

- *Informacją pełną* – nazywamy informację kompletną, uzupełnioną o znajomość przez każdego z graczy funkcji wypłat oraz stanu wszystkich pozostałych uczestników gry na każdym jej etapie.
- **Determinizm** – gry deterministyczne to gry pozbawione elementów losowych (rzut kostką, monetą, układ kart) w przeciwieństwie do gier niedeterministycznych (losowych), w których taki element ma miejsce.
- **Skończoność** – gry skończone to takie, w których gracz posiada skońzoną liczbę strategii czystych¹. Przykładem gry skończonej są *Szachy*. Gry nieskończone dzielą się na takie w których gracz posiada przeliczalną ilość strategii, bądź na takie gdzie zbiór strategii jest większej mocy.
- **Etapowość**
 - *Gry jednoetapowe* – to takie, w których na każdego z graczy przypada jedno posunięcia po którym następuje wypłata.
 - *Gry wieloetapowe* – mogą składać się ze skończonej albo nieskończonej liczby posunięć. W grach o skończonej liczbie posunięć każda strategia gracza musi składać się ze skończonej liczby posunięć. W grach towarzyskich często wprowadzane są dodatkowe zasady pozwalające na spełnienie tego warunku. Przykładowo *Szachy* kończą się remisem jeśli nastąpi trzykrotne powtórzenie tej samej pozycji. Istnieją gry o nieskończonej liczbie posunięć, w których gracz również może posiadać strategię wygrywającą, przykładem jest *Gra Banacha-Mazura*. GRY o nieskończonej liczbie ruchów są z powodzeniem stosowane w różnych dziedzinach matematyki, takich jak teoria mnogości bądź topologia. Jednakże są to wyimaginowane procesy i nie bardzo można w nie grać w klasycznym tego słowa znaczeniu.
- **Równoległość**
 - *Gry równolegle* – to takie, w których gracze równocześnie podejmują decyzję. Są to gry z niepełną informacją, ponieważ żaden z graczy nie zna decyzji przeciwników.

¹Strategia czysta, to taka której wyboru gracz dokonuje z prawdopodobieństwem równym 1. Jest ona szczególnym przypadkiem strategii mieszanej, w której wybór postaci strategii dokonywany jest względem rozkładu prawdopodobieństwa przypisanego do zbioru strategii czystych gracza.

- *Gry sekwencyjne* – to takie, gdzie gracze podejmują decyzje kolejno. Stąd decyzja podjęta przez jednego z graczy ma wpływ na decyzję przeciwnika. Są to przykładowo: *Warcaby*, *Szachy*, *Go*.
- **Złożoność przestrzeni stanów** - jest to liczba wszystkich możliwych odmiennych pozycji w grze. Zakłada się, że do rozwiązywania gier o dużej liczbie pozycji najskuteczniejsze są metody przeszukujące, bazujące na wiedzy (ang. *knowledge-based methods*). Wiele z nich szczegółowo opisanych jest w sekcjach 3.3.1-23.
- **Złożoność drzewa gry** – jest to liczba wszystkich możliwych sposobów rozegrania gry. Zakłada się, że do rozwiązywania gier o dużej złożoności drzewa gry najskuteczniejsze są metody siłowe (ang. *brute-force methods*) polegające na przeglądzie całej przestrzeni stanów gry. Metodę taką opisano w sekcji 3.2.
- **Złożoność obliczeniowa** - grę możemy rozważać jako problem obliczeniowy zdefiniowany następująco: „Jaka jest wypłata graczy, dla danej sytuacji w grze, jeżeli każdy z nich realizuje swoją optymalną strategię?” Dla każdej gry o ścisłe określonej złożoności przestrzeni stanów, problem ten jest teoretycznie rozwiązywalny w czasie stałym. Dlatego, aby gry traktowane jako problemy decyzyjne¹, przypisać do różnych klas złożoności obliczeniowej, należy każdą z nich uogólnić do przestrzeni stanów o dowolnej złożoności. Przykładowo, dla gier planszowych należy nieznacznie zmienić zasady gry, tak aby możliwe było rozegranie partii na planszy o dowolnych rozmiarach i aby dla szczególnego przypadku, reprezentującego oryginalną grę jej zasady nie uległy zmianie. Dla gry typu *Kółko o Krzyżyk* jest to bardzo proste i takie uogólnienie tworzy rodzinę ***mnk-Gier***. Są to gry, w których na planszy o wymiarach $(n \times m)$ należy ułożyć ciąg składający się z k pionów gracza. Takie uogólnienie dla szachów może się wydawać bardziej skomplikowane, możemy jednak założyć, że nie interesuje nas wyłącznie wynik rozgrywki z pozycji startowej a z każdej dowolnej kombinacji pionków i figur na szachownicy.

Stosując powyższe założenia można wyodrębnić pięć grup złożoności obliczeniowej wraz z zawierającymi się w nich gromadzącymi traktowanymi jako problemy w nich zupełnie²:

¹ Problem decyzyjny to pytanie sformułowane w taki sposób, że można na nie udzielić tylko jednej z dwóch możliwych odpowiedzi: tak bądź nie.

² Problemy zupełnie to problemy najtrudniejsze w danej klasie złożoności obliczeniowej, charakteryzują się tym, że każdy inny problem należący do tej klasy bądź do innych klas zawierających

- **P** – *Nim*.
 - **NP** – *Saper, Tetris*.
 - **PSPACE** – *Geografia, Kółko i Krzyżyk, Go-Moku, Renju, Hex, Othello*.
 - **EXPTIME** – *Warcaby, Szachy, Go*.
 - **EXPSPACE** – *G1*¹.
- **Zbieżność** – istnieją gry o charakterze typowo zbieżnym. Charakteryzują się one taką cechą, że podczas trwania rozgrywki maleje liczba dostępnych stanów gry. Przykładowo: *Szachy, Warcaby* są to gry, które rozpoczynają się z dużą liczbą bierek na planszy a kończą z niewielką. W tego typu grach występuje bicie bierek. Gry rozbieżne to takie, w których dostępna przestrzeń stanów rośnie podczas rozgrywki. Przykładowo: *Go, Go-Moku, Renju* są grami rozbieżnymi. W tego typu grach pozycja początkowa najczęściej jest pusta plansza, która zapełnia się bierkami podczas trwania gry. Istnieją również gry, które mają naturę rozbieżno-zbieżną, jak *Młynek* oraz takie, w których dostępna przestrzeń stanów nie ulega większym zmianom podczas rozgrywki jak *Skoczki*.
 - **Popularność** – na popularność gry mają wpływ takie czynniki jak:
 - liczba graczy na świecie,
 - liczba przeprowadzonych rozgrywek.
 - **Trudność** – gry trudne to takie w których doświadczenie gracza ma wpływ na wartość wypłaty bądź na prawdopodobieństwo wygranej w grach, w których wypłata jest ograniczona do wartości ze zbioru: $\{wygrana, przegrana, remis\}$. Przykładowo gry trudne to *Szachy* lub *Go* ale również gry niedeterministyczne jak *Backgamon* bądź *Brydż*. Gry trywialne, gdzie doświadczenie gracza nie wpływa na wynik to proste gry w karty, grywane przez młodsze dzieci albo *Ruletkę* grywaną przez dzieci starsze.

Teraz można już zdefiniować zbiór gier które będą przedmiotem badań niniejszej pracy. Są to gry logiczne, dwóch graczy, deterministyczne, skończone lub o możliwej nieskończonej liczbie posunięć lecz skończonej liczbie możliwych stanów gry, z pełną informacją, o zerowej sumie wypłat, popularne, a zarazem nietrywialne, uff...

się w niej można do niego wielomianowo zredukować.

¹Przykład i dowód na EXPSPACE-zupełność gry *G1* można znaleźć w [44].

2.2. Rozwiązać grę – co to znaczy?

W 1913 roku niemiecki matematyk Ernst Zermelo (1871–1953) przedstawił w niemieckim czasopiśmie szachowym dowód pierwszego formalnego twierdzenia z dziedziny teorii gier. Koncentrując się na analizie gier deterministycznych dla dwóch graczy o sumie zerowej, rozważał te gry, w których występuje skończona przestrzeń stanów, ale w których liczba posunięć graczy może być nieskończona. W rezultacie tych prac sformułował dwa problemy: [55]

- Co oznacza dla gracza być na „zwycięskiej” pozycji?
- Jeśli gracz znajduje się na pozycji zwycięskiej, czy liczba ruchów do osiągnięcia wygranej jest ścisłe określona?

Rozważając pierwszą kwestię stwierdził, że koniecznym i wystarczającym warunkiem istnienia pozycji wygranej dla gracza A , jest istnienie niepustego zbioru, zawierającego wszystkie możliwe sekwencje posunięć, które:

- rozpoczynają się od rozważanej pozycji,
- niezależnie od posunięć przeciwnika B , gracz A zawsze osiągnie pozycję reprezentującą jego wygraną¹.

Nazwijmy ten zbiór zbiorem strategii wygrywających gracza A . Jeśli jest on pusty, gracz A w najlepszym wypadku może zremisować.

Zermelo rozważając drugą kwestię, zdefiniował zbiór zawierający wszystkie możliwe sekwencje posunięć gracza A , opóźniające jego przegraną w nieskończoność. Takie nieskończone opóźnianie partii przez gracza A , prowadzi do remisu. Nazwijmy ten zbiór zbiorem strategii remisujących. Jeśli on również będzie pusty, to gracz A może opóźniać swoją przegraną tylko w skończonej liczbie posunięć. Z tego wynika fakt, że gracz B grając optymalnie musi w skończonej liczbie ruchów wygrać grę.

Reasumując, jeśli gracz znajduje się na pozycji zwycięskiej to posiada niepusty zbiór strategii wygrywających i w skończonej liczbie kroków musi wygrać zakładając, że realizuje strategię z tego zbioru.

Następnie, rozważając drugi z postawionych problemów Zermelo stwierdził, że gracz A , który znajduje się na zwycięskiej pozycji osiągnie wygraną w nie większej liczbie posunięć niż liczba możliwych stanów gry (złożoność przestrzeni stanów).

¹Musimy rozróżnić pozycję zwycięską od pozycji reprezentującej wygraną. Z pozycji zwycięskiej możemy wykonać przynajmniej jedno kolejne posunięcie prowadzące do kolejnej pozycji zwycięskiej, natomiast z pozycji reprezentującą wygraną nie możemy wykonać już żadnego posunięcia, gra uległa zakończeniu i gracz wygrał rozgrywkę.

Założymy, że gracz A może osiągnąć wygraną w sekwencji ruchów o liczności przekraczającej liczbę stanów gry. Wtedy przynajmniej jedna pozycja wygrywająca gracza A musiała wystąpić dwukrotnie. Wobec tego gracz A może już z pierwszego wystąpienia tej pozycji wykonać posunięcia, które wykonał po drugim jej wystąpieniu.

Reasumując, albo jeden z graczy A i B posiada strategię wygrywającą, albo też obaj mają strategie prowadzące do remisu, gdzie remis rozumiemy jako zakończenie gry stanem remisowym bądź jej kontynuowanie w nieskończoność.

Odpowiadając na pytanie postawione w tytule rozwiązywanie gry należącej do zbioru, który zdefiniowałem pod koniec sekcji 2.1 oznacza, że jesteśmy w stanie przewidzieć wynik rozgrywki dla graczy wykonujących zawsze optymalne dla nich posunięcia (realizujących optymalne strategie).

Mówiąc o rozwiązyaniu gry można mieć na myśli przynajmniej trzy sytuacje:

- **rozwiązańe ultra-lekkie** – zaproponowane przez Paula Colley'a. Oznacza, że określona jest wartość gry dla pozycji startowej. Jeżeli gracze będą grali optymalnie to z góry wiadomo jaki będzie wynik rozgrywki. Jednakże nie musi być znana jakakolwiek strategia pozwalająca na osiągnięcie tej wartości przez któregośkolwiek z graczy.
- **rozwiązańe lekkie** – zaproponowane również przez Paula Colley'a. Oznacza, że określona jest wartość gry dla pozycji startowej, oraz znana jest strategia pozwalająca na osiągnięcie tej wartości¹. Przykładowo jeśli pozycja startowa dla jednego z graczy jest wygrana to zna on sekwencję posunięć prowadzącą do wygranej, niezależnie od posunięć wykonywanych przez przeciwnika.
- **rozwiązańe silne** – zaproponowane przez Donaldha Michiego. Oznacza, że dla każdej legalnej pozycji² występującej w grze określona jest jej wartość oraz strategia pozwalająca na osiągnięcie tej wartości³. Przykładowo jeśli gracze będą wykonywać ruchy nieoptymalne (zmieniające sytuację na planszy bądź przedłużające rozgrywkę) to dalej dla każdej z osiągniętych pozycji znana jest jej wartość i strategia optymalna.

¹Jeżeli pozycja startowa jest wygrana to znana jest strategia optymalna gracza wygrywającego.

Jeśli pozycja startowa jest remisowa to znane są strategie optymalne obojga graczy.

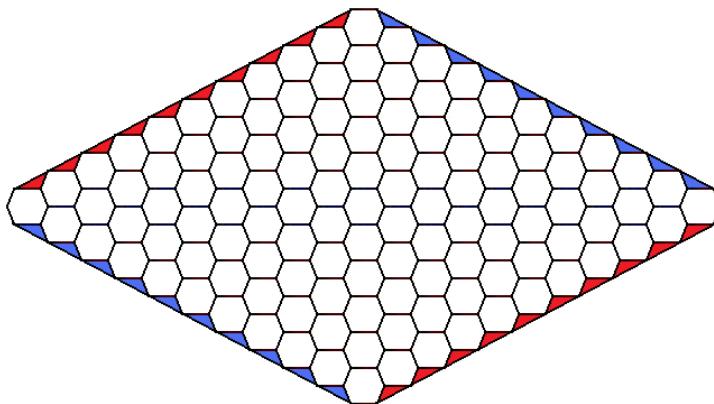
²Pozycja legalna to taka, którą można osiągnąć z pozycji startowej gry poprzez sekwencję zgodnych z regułami tej gry posunięć.

³Dla każdej pozycji znana jest (analogicznie jak w przypadku rozwiązania lekkiego) albo optymalna strategia gracza wygrywającego albo w przypadku remisu, strategie optymalne obydwu graczy.

Łatwo zauważyc, że rozwiązańe silne zawiera rozwiązańe lekkie a rozwiązańe lekkie zawiera rozwiązańe ultra-lekkie. Z tego wynika, że rozwiązańe silne również zawiera rozwiązańe ultra-lekkie.

2.3. Gry rozwiązane

Pierwszą nietrywialną grą, która doczekała się rozwiązań jest *Hex*. Jest to gra o bardzo prostych zasadach¹, której rozgrywka toczy się na heksagonalnej planszy o wymiarach (11×11) rys. 2.1.



RYSUNEK 2.1. Plansza do gry w *Hex'a* (11×11)

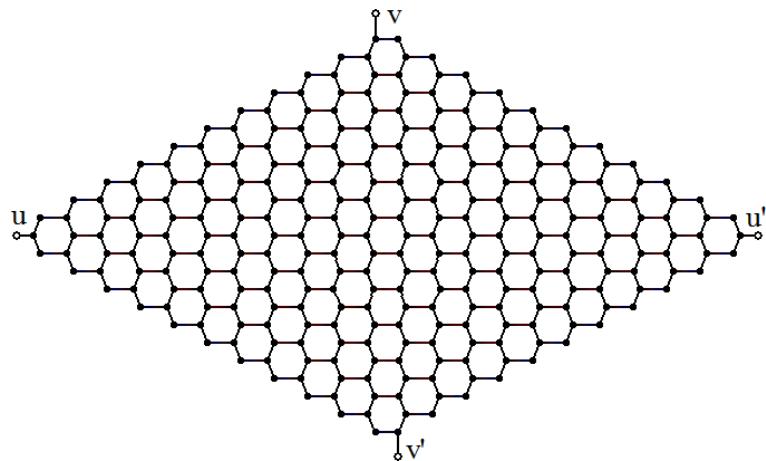
Ultra-lekkie rozwiązań tej gry podał jej twórca John Nash². Należy zauważyc, że topologia planszy jest tak dobrana aby niemożliwe było przeprowadzenie rozgrywki zakończonej remisem. Dowód tej własności przeprowadził David Gale [6], reprezentując planszę do gry w formie grafu z dodatkowymi czterema wierzchołkami: u, u', v, v' , separującymi cztery boki planszy – rys. 2.2. Algorytm odkrywania zbioru wygrywającego na całkowicie zapełnionej planszy³ jest następujący:

1. Zaczynając od wierzchołka u przejdź do sąsiedniego wierzchołka.
2. Przejdź z bieżącego wierzchołka do sąsiedniego poprzez krawędź oddzielającą dwa pola należące do różnych graczy (pomiędzy dwoma różnymi kolorami) ale nie poprzez krawędź, którą trafiłeś do bieżącego wierzchołka.

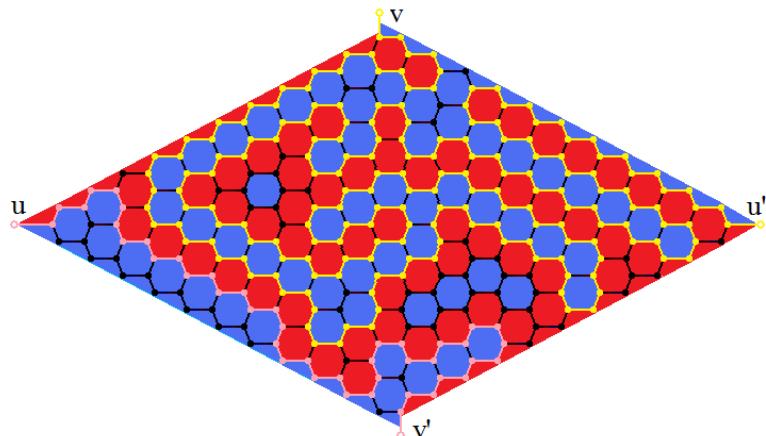
¹Szczegółowe zasady gry *Hex* opisano w zamieszczono B.11.

²Jako pierwszy grę wynalazł duński matematyk Piet Hein w 1942 roku. Nash wymyślił tę grę niezależnie od Hein'a w 1948.

³Zakładamy, że rozgrywka w Hex'ie dla kompletnie zapełnionej planszy musi się zakończyć wygraną jednego z graczy. W praktyce koniec rozgrywki następuje dużo wcześniej.

RYSUNEK 2.2. Plansza do *Hex'a*, przedstawiona w formie grafu

3. Powtarzaj krok 2, aż osiągniesz jeden z dwóch wierzchołków: u' , v , v' .
4. Powtórz kroki 1 – 3 zaczynając od wierzchołka v a kończąc w jednym z wierzchołków: u , u' , v' .

RYSUNEK 2.3. Zapełniona plansza do *Hex'a*, przedstawiona w formie grafu

Przykład działania tego algorytmu przedstawia rys. 2.3. Wytyczy on zawsze dwie unikalne ścieżki pomiędzy czterema dodanymi wierzchołkami. Wynika to z faktu, że podczas wykonywania algorytmu będąc w jakimś wierzchołku w , dotarliśmy do niego poprzez pewną krawędź e . Ponieważ krawędź e musi oddzielać dwa pola o różnym kolorze a wierzchołek w zawsze znajduje się pomiędzy trzema polami¹, kolor trzeciego pola sąsiadującego z wierzchołkiem w musi być zatem identyczny z jednym

¹Wliczamy tutaj również pola brzegowe planszy.

z dwóch wierzchołków przylegających do e i zarazem przeciwny do jednego z nich. Teraz mogą wystąpić dwa przypadki:

1. Jeśli wierzchołek jest stopnia 3 to jest incydentny z trzema krawędziami. Jedna z nich to e , natomiast z dwóch pozostałych, jedna musi przechodzić pomiędzy polami tego samego koloru a druga pomiędzy polami o różnych kolorach i tą krawędzią podąży algorytm.
2. Jeśli wierzchołek w jest stopnia 2 to jest incydentny tylko z dwoma krawędziami (przypadek ten może wystąpić wyłącznie na brzegu planszy). Wtedy algorytm ma do wyboru tylko jedną krawędź i na pewno jest ona poprawna. Wynika to z faktu, że dwa pola reprezentujące brzeg planszy zawsze są tego samego koloru i przeciwnego do graniczącego z nimi wierzchołka. Gdyby było inaczej to algorytm nie przeszedłby krawędzią e znajdującą się pomiędzy brzegiem planszy, a wierzchołkiem tego samego koloru.

Jak widać algorytm nie może się zatrzymać gdzieś w środku planszy. Musi się zatem zakończyć wykonywać w jednym z trzech spośród czterech dodanych wierzchołków. Algorytm również nie może się zapętlić ponieważ liczba wierzchołków jest skończona a każdy wierzchołek może zostać odwiedzony tylko raz. Wynika to z twierdzenia 2.1.

Twierdzenie 2.1. *Każdy skończony graf, składający się z wierzchołków co najwyżej stopnia 2, jest sumą rozłącznych podgrafów, z których każdy może być:*

- *wierzchołkiem izolowanym,*
- *prostym cyklem,*
- *prostą ścieżką.*

Ponieważ nasz algorytm wykorzystuje co najwyżej dwie krawędzie każdego z odwiedzanych wierzchołków, powyższe twierdzenie można tutaj z powodzeniem zastosować. Łatwo to sobie wyobrazić wyrzucając z grafu te krawędzie, które znajdują się pomiędzy polami o identycznej barwie. Pozostają wtedy tylko cztery wierzchołki stopnia 1. Są to wierzchołki u, u', v, v' , a że każda ścieżka posiada w sobie dokładnie dwa takie wierzchołki, początkowy i końcowy, w naszym grafie istnieją dokładnie dwie, znalezione przez nasz algorytm ścieżki. Ponieważ nie mogą się one przecinać, możliwe są tylko dwa przypadki:

- Istnieją dwie ścieżki: u, \dots, v i v', \dots, u' .

- Istnieją dwie ścieżki: u, \dots, v' i v, \dots, u' .

W pierwszym przypadku obie ścieżki separują¹ obydwa brzegi gracza czerwonego, a zbiór wygrywający gracza niebieskiego znajduje się pomiędzy nimi i przylega do każdej z nich. W drugim przypadku występująca sytuacja jest analogiczna dla kolejnego (niebieskiego) gracza i jego pozycja musi być wygrywająca.

Jak widać możliwe są tylko dwie sytuacje i każda z nich to wygrana jednego z graczy, remis jest więc niemożliwy.

W 1949 roku Nash wykorzystując argument z kradzenia strategii [7, 8] własnego autorstwa wykazał, że gracz rozpoczęty rozgrywkę, nazwijmy go A , musi posiadać strategię wygrywającą. Poniżej zamieszczono interpretację wspomnianego dowodu przez autora tej pracy.

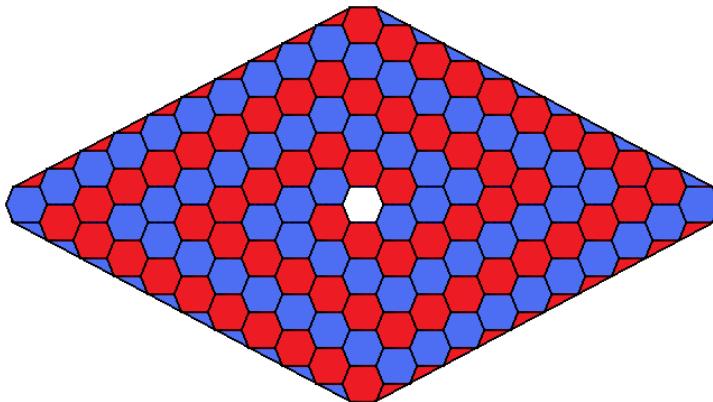
Załóżmy nie wprost, że przeciwnik, nazwijmy go B , posiada strategię wygrywającą. Gracz A może postawić swój pierwszy pionek gdziekolwiek na planszy i o nim „zapomnieć”, a następnie wykradając strategię przeciwnika B , pozwolić mu „rozpocząć grę”. Jeśli skradziona strategia jest wygrywającą dla gracza nie rozpoczynającego grę to gracz A powinien wygrać. Zakładamy bowiem, że pierwszy ruch się nie odbył. Można tak założyć, ponieważ zasad gry wynika, że dodatkowy pionek koloru gracza A nie może mu w żaden sposób przeszkodzić. Ponadto, jeśli gracz A , posługując się strategią przeciwnika B , będzie musiał postawić pionek na polu wcześniej przez niego zajętym to możemy założyć, że wykonał już to posunięcie, a żeby być w zgodzie z przepisami gry, stawia własny pionek na dowolnym jeszcze nie zajętym polu.

Topologia planszy do gry gwarantuje, że dopóki gra nie uległa zakończeniu musi istnieć przynajmniej jedno takie wolne pole. Na rys. 2.4 widzimy niezakończoną rozgrywkę z tylko jednym polem wolnym. Dowolny z graczy, który postawi na nim swój pionek zakończy grę, jednocześnie wygrywając tę partię. Oznacza to, że istnienie strategii wygrywającej dla gracza B , jako nierozpoczynającego, prowadzi do sprzeczności. Skoro remis jest również niemożliwy, gracz A musi posiadać strategię wygrywającą.

W *Hex'ie konkurencyjnym*¹ strategię wygrywającą posiada gracz decydujący o wyborze swojego koloru, czyli drugi.

¹zbiór wygrywający gracza, nie może przecinać żadnej ze znalezionych ścieżek, ponieważ własnością ścieżki jest to, że zawsze przechodzi ona pomiędzy polami różnego koloru, a zbiór wygrywający musi być ciągły i jednokolorowy.

¹Szczegółowe zasady *Hex'a konkurencyjnego* zamieszczone w dodatku B.12.



RYSUNEK 2.4. Przykład nie zakończonej rozgrywki w *Hex'a* z jedynym wolnym polem

Argument z kradzenia strategii można stosować w wielu grach z rodziny gier połączeniowych². Jest on poprawny, jeżeli udowodnimy, że pierwszy „dodatkowy” ruch gracza nie może mu przeszkodzić podczas trwania rozgrywki. Przykładowo gry: *Kółko i Krzyżyk*, *Czwórki*, *Go-Moku wolne* oraz *Hex* są grami, w których gracz nierozpoczynający, nie może posiadać strategii wygrywającej.

Należy zauważyć, że o ile w przypadku *Go-moku wolnego* założenie to jest spełnione, to dla klasycznego *Go-moku* ustwiony wcześniej pionek może przeszkodzić w osiągnięciu wygranej, przekształcając zwycięską piątkę³ w neutralną szóstkę. Jednak w każdym z pozostałych przypadków graczowi nie opłaca się rezygnować z posunięcia. Dla wariantu *Renju* sytuacja jeszcze bardziej się komplikuje, ponieważ dodatkowy pion nie tylko może przeszkodzić w rozgrywce, ale i bezpośrednio zadecydować o jej wyniku. Ułożenie przez gracza grającego czarnymi wypomnianej szóstki oznacza jego przegrana.

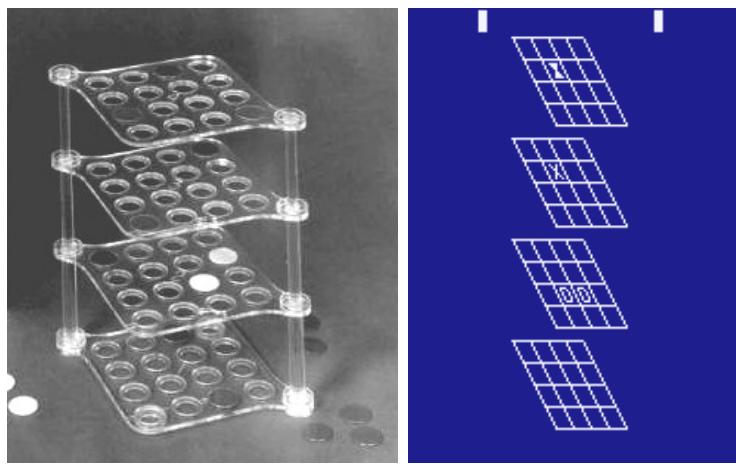
Dla każdej nietrywialnej gry znalezienie strategii wymagało dużej liczby obliczeń oraz zastosowania zaawansowanych technik przeszukiwania drzewa gry. Pierwszą grą, która doczekała się lekkiego rozwiązania był *Qubic*¹. Jest to gra rozgrywana na trójwymiarowej planszy – rys. 2.5.

W 1980 roku Oren Patashnik [39] udowodnił, że rozpoczynający grę posiada strategię wygrywającą. Potwierdził w ten sposób poprawność użycia opisanego wcześniej

²Gry połączeniowe to takie w których gracz wygrywa tworząc pewną, określona strukturę poprzez połączenie (ustawienie na sąsiadujących polach) swoich pionów.

³Pięć i tylko pięć pionów koloru gracza w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie).

¹Szczegółowe zasady gry *Qubic* zamieszczone w dodatku B.22.



(a) jako trójwymiarowa plansza do gry (b) jako gra na Atarii

RYSUNEK 2.5. Różne wcielenia tej samej gry *Qubic*

argumentu z kradzenia strategii w przypadku gier połączeniowych. Patashnik przeprowadził obliczenia pracując w laboratoriach firmy Bell. Jego program potrzebował 1500 godzin pracy komputera, a znaleziona strategia wygrywająca składała się z 2928 strategicznych posunięć.

Kolejna rozwiązana gra również należy do gier połączeniowych. W Polsce znana jest pod nazwą *Czwórki*². Gra przedstawiona została na rys. 2.6. Posiada ona pionową orientację co powoduje, że prawo grawitacji ogrywa w niej istotną rolę polegającą na zmuszaniu pionków przeciwnika do osiągania najniższych, nie zajętych pozycji. Została ona lekko rozwiązana niezależnie przez dwie osoby. Autorem pierwszego rozwiązania był w 1988 roku James D. Allen [1], a zaledwie kilka tygodni później dokonał tego samego Victor Allis [2] przy pomocy programu „Victor” i kombinacji metod: „conspiracy-number search”, „search tables” i „depth-first search”. Obydwaj zgodnie wykazali, że gracz rozpoczynający posiada strategię wygrywającą, a co za tym idzie – ponownie potwierdzili regułę, że wiele gier z rodziny połączeniowych nie posiada strategii wygrywającej dla gracza nierozpoczynającego.

W 1990 roku postawiono pewne hipotezy co do rozgrywanych wówczas, na dwóch pierwszych olimpiadach komputerowych, gier. Dotyczyły one siły programów komputerowych grających w te gry. Przewidywania te były datowane na 2000 rok, czyli

²Szczegółowe zasady gry *Czwórki* zamieszczone w dodatku B.3.

RYSUNEK 2.6. Gra *Czwórki*

dotyczyły kolejnej dekady. [23, 33]. Jak na tamte czasy były to były to bardzo śmiałe a zarazem trudne ustalenia. Dla porównania zobaczymy ile prognoz, odnośnie różnych wersji końca świata bądź katastrofy komputerowej, datowanych na rok 2000 nie zostało potwierdzonych w praktyce.

TABLICA 2.1. Hipotezy z 1990 na 2000 rok, dotyczące siły programów, grających w gry z olimpiad komputerowych

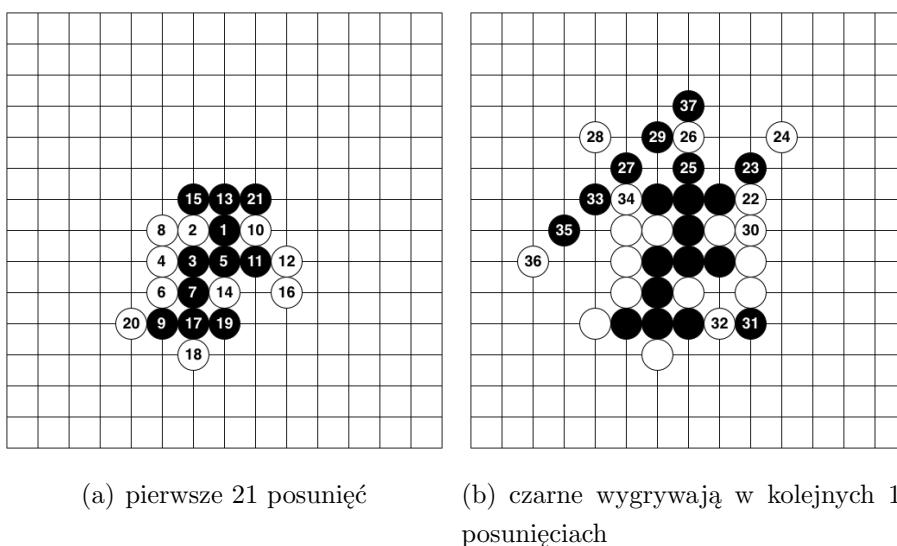
<i>rozwiązywany</i>	<i>ponad mistrzowski</i>	<i>mistrz świata</i>	<i>mistrz</i>	<i>amator</i>
Czwórki	Warcaby angielskie	Szachy	Go (9×9)	Go
Qubic	Renju	Warcaby polskie	Szachy chińskie	
Młynek	Othello		Brydż	
Go-Moku	Scrabble			
Awari	Backgammon			

Dzisiaj możemy stwierdzić, że wyniki ówczesnych przewidywań, które przedstawiono w tab. 2.1.¹ w dużym stopniu okazały się słuszne. Jedynym wyjątkiem dotyczącym rozwiązywanych gier była *Awari*, która do 2002 roku opierała się rozwiązyaniu. Co ciekawe *Qubic* była grą rozwiązywaną już w 1980 roku ale twórcy tego przypuszczenia nie byli jeszcze świadomi istnienia jej rozwiązania.

Przewidywania co do pozostałych nieroziwiązanych gier, również były bliskie rzeczywistości dwutysięcznego roku. Jednak tutaj granica tej oceny staje się coraz bardziej subiektywna, a metody wykorzystywane w tych programach mniej adekwatne do tematyki niniejszej pracy.

¹Poziom *ponad mistrzowski* to taki w którym gra nie jest rozwiązana, ale program zawsze wygra z człowiekiem.

W 1990 roku Victor Allis i Patrick Schoo w błogiej nieświadomości istniejącego od dziesięciu lat rozwiązywania pracowali nad grą *Qubic*. Pomimo, że zostali o tym poinformowani u schyłku swojej pracy postanowili zakończyć dzieło. Potraktowali ten problem jako poligon doświadczalny dla przetestowania nowych metod: „Threat-space search” i „Proof-number search”, które w 1992 roku Victor Allis z powodzeniem zastosował podczas rozwiązywania gry *Go-Moku* [3, 5].



RYSUNEK 2.7. Przykładowa rozgrywka w grze *Go-Moku*

Wykorzystując wcześniej wspomniane algorytmy w programie „Victoria” udowodnił, że zarówno dla wolnego jak i standardowego *Go-Moku*¹, gracz rozpoczynający posiada strategię wygrywającą.

Gra charakteryzuje się ogromną złożonością przestrzeni stanów $\approx 10^{105}$, jednak największym jej problemem był bardzo duży współczynnik rozgałęzienia, średnio przekraczający 200 ruchów. Do jego redukcji zastosowano „heurystykę pustych ruchów”. Pozwoliła ona na redukcję możliwej liczby ruchów do kilkudziesięciu.

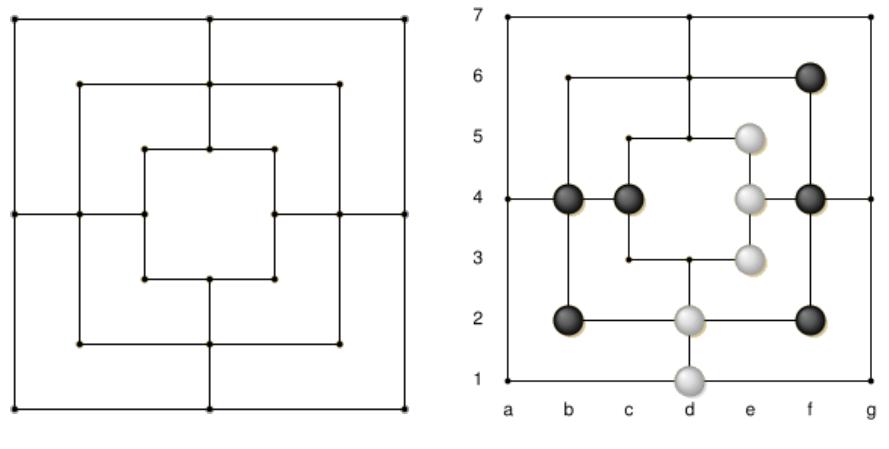
„Heurystyka pustych ruchów” opiera się na założeniu, że gracz czarny posiada strategię wygrywającą, co w przypadku *Go-Moku wolnego* jest prawdą, a w przypadku klasycznej wersji musimy zawierzyć naszej intuicji, że również tak jest. Teraz aby udowodnić, że gracz czarny na pewno wygra wystarczy, by w każdej możliwej pozycji znać tylko jeden ruch prowadzący do kolejnej pozycji zwycięskiej. Natomiast aby być pewnym, że przeciwnik nie ma możliwości obrony, należy przeanalizować

¹Szczegółowe zasady obydwu wariantów *Go-Moku* zamieszczono w dodatkach B.7 i B.8.

zować wszystkie jego posunięcia. Dlatego, początkowo w odpowiedzi na każdy ruch czarnego gracza wykonujemy „puste posunięcie” gracza białego w celu szybkiego znalezienia optymalnych posunięć, składających się na sekwencję wygrywającą gracza czarnego. Ocena tych sekwencji jest wykonywana poprzez prosty algorytm heurystyczny, ścisłe powiązany z zasadami gry. Dopiero później wykonywana jest szczegółowa analiza wyselekcjonowanych, najbardziej rokujących posunięć.

„Victoria” na czwartej Olimpiadzie komputerowej w Londynie, w 1992 roku, uczestnicząc w turnieju *Go-moku* wygrała wszystkie rozgrywki grając czarnymi¹. Natomiast grając białymi udało jej się wygrać połowę partii. To jej zapewniło zdobycie złotego medalu.

Rok później rozwiązyano grę *Młynek*². Planszę oraz jedną z możliwych pozycji tej gry przedstawiono na rys. 2.8. Ta gra znana od co najmniej trzech tysięcy lat o angielskiej nazwie *Nine Men’s Morris* została rozwiązana przez Ralpha Gasser’a [19], który udowodnił, że przy optymalnej grze obydwu graczy kończy się ona remisem.



(a) pusta plansza

(b) pozycja wygrywająca dla białych

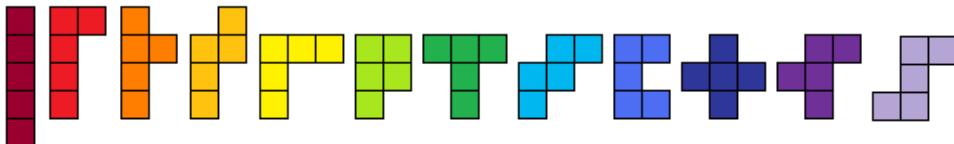
RYSUNEK 2.8. Gra *Młynek*

Jest to gra nietypowa, ponieważ w początkowej fazie posiada naturę rozbieżną, która w drugiej części gry zamienia się w zbieżną. Z tego powodu do jej rozwiązania po raz pierwszy zastosowano kombinację dwóch technik. W początkowej części gry, w której liczba ruchów jest stała i wynosi 18 (od liczby pionków), wykorzystano

¹Przypominam, że grający czarnymi w *Go-moku* rozpoczynają oraz posiadają strategię wygrywającą.

²Szczegółowe zasady gry *Młynek* zamieszczone w dodatku B.16.

metodę szukania „18-ply α - β ”. W końcowej części gry gdzie liczba ruchów zależy od wykonywanych przez graczy posunięć, użycie analizy przeszukiwania wstecz. Pozwoliła ona na skonstruowanie bazy danych, zawierającej wszystkie pozycje i wartości tej części gry. Zatem rozwiązanie całej gry jest rozwiązaniem lekkim, jednakże jej druga połowa została silnie rozwiązana.



RYSUNEK 2.9. Pionki do gry w *Pentomino*

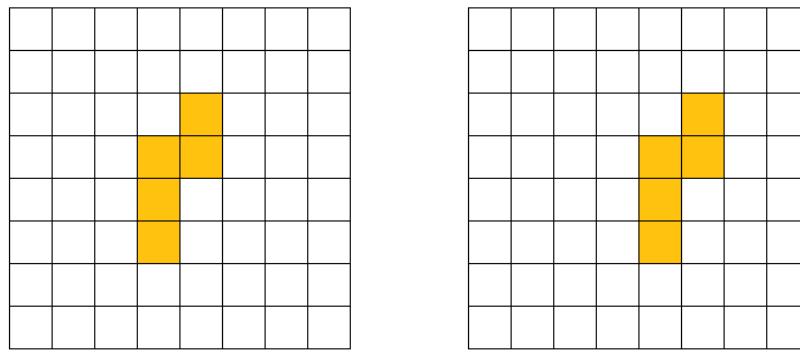
W 1995 John Tromp opracował silne rozwiązanie gry *Czwórki*. Obliczenia przeprowadził z wykorzystaniem serwera Sun i stacji roboczej SGI, w holenderskim „Centrum Wiskunde & Informatica” (CWI). Trwały one 40.000 godzin. Program w postaci appletu java dostępny jest na jego stronie [60].

W 1996 roku Hilerie Orman wykorzystując około dwóch tygodni czasu pracy 64-bitowego komputera z procesorem DEC Alpha 175 MHz [37] znalazł początkową zwycięską pozycję w grze *Pentomino*¹. Pionki używane w tej grze przedstawiono na rys. 2.9. Jest to jedna z wielu możliwych początkowych pozycji zwycięskich. W *Pentominie* występuje 2308 możliwości otwarcia, a po uwzględnieniu symetrii 296. Po wykonaniu pierwszego posunięcia pozostaje od 1181 do 2000 możliwości odpowiedzi przeciwnika. Przeciętna rozgrywka składa się z dziesięciu posunięć, najkrótsza z pięciu, a najdłuższa z dwunastu.

Na rys. 2.10(a) przedstawiono znalezioną startową pozycję wygrywającą, dla której występuje 1197 odpowiedzi przeciwnika. Inna wygrywająca pozycja sugerowana przez Solomona W. Golomb'a została przedstawiona na rys. 2.10(b).

Komputer przeszukujący drzewo gry był napisany w języku C i używa prostej metody „przeszukiwania z nawrotami”. Posunięcia i stany gry zapamiętywane były w postaci wektorów bitowych. Algorytm na starcie (poziom 1) posiadał listę wszystkich możliwych legalnych ruchów. Po każdym posunięciu w grze lista ta ulegała redukcji poprzez usunięcie już niemożliwych do wykonania ruchów. Jeśli wszystkie możliwe posunięcia na głębokości n były przegrywające to zwracana była wartość

¹Szczegółowe zasady gry *Pentomino* zamieszczono w dodatku B.19.



(a) pierwsza udowodniona przez Orman'a
 (b) sugerowana przez Golomb'a

RYSUNEK 2.10. Startowe pozycje w *Pentomino* zapewniające wygraną rozpoczynającemu

wygrywającą do poziomu $n - 1$. W przeciwnym wypadku, jeśli jakikolwiek ruch był wygrywający, zawracana wartość była przegrywającą.

Kolejną rozwiązaną grą jest *Quarto*¹. Została ona wymyślona przez Blaise Müller'a, a w 1993 roku ogłoszona, według towarzystwa MENSA, najlepszą grą roku. Planszę do gry wraz z pionkami można zobaczyć na rys. 2.11. W 1998 roku Luc Goossens



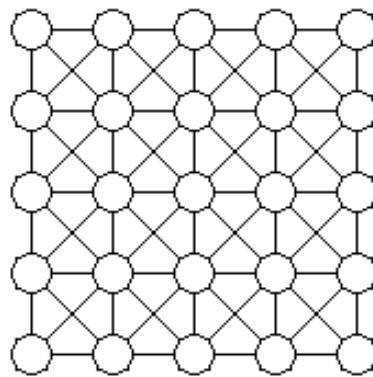
RYSUNEK 2.11. Plansza i pionki do gry *Quarto*

wykazał, że wersja *Twist* tej gry przy optymalnej rozgrywce zawsze kończy się remisem. Natomiast w 2003 roku Arthur Holshouser i Harold Reiter dla wersji klasy-

¹Szczegółowe zasady gry *Quarto* i *Quarto twist* zamieszczone w dodatkach B.20 i B.21.

cznej pokazali strategię wygrywającą drugiego gracza [24].

Kolejną grą, która podzieliła los swoich poprzedniczek była współczesna, opracowana w 1937 roku przez Johna Scarne [61] gra *Teeko*², której plansza do gry została przedstawiona na rys. 2.12.



RYSUNEK 2.12. Plansza do gry *Teeko*

W 1998 roku Guy Stelle wykazał, że w zależności od wariantu (których jest 16) gracz rozpoczynający posiada strategię wygrywającą albo przynajmniej remisową. Dla standardowych zasad rozgrywka optymalnie prowadzona przez obydwu graczy zawsze prowadzi do remisu.

Istnieje bardzo stara i bogata rodzina gier zwana grami *Mankala*. Larry Russ [46] podaje ponad sto tradycyjnych odmian tej gry. Są one rozgrywane na planszy z zagłębieniami, które służą do przechowywania różnej liczby pionków podczas trwania rozgrywki. Pionki często występują pod postacią: kamieni, muszli bądź nasion, od których pojedynczy ruch nazwano sianiem. Przykładową planszę do gry pokazano na rys. 2.13.

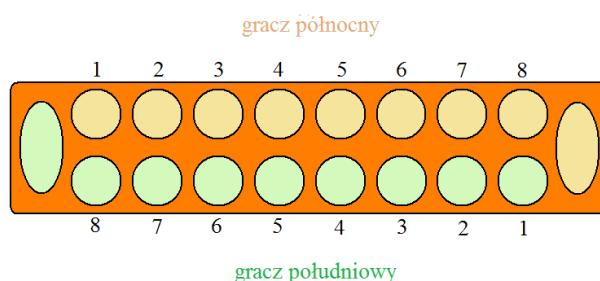
Pierwszą z gier Mankala, która doczekała się rozwiązania, był *Dakon*¹.

Co ciekawe otwarcie wygrywające dla tej gry zostało podane przez graczy z Malediw, gdzie jest ona znana pod nazwą *Ohvalhu* i grana jest zgodnie z ruchem wskazówek zegara¹. Otwarcie wygrywające to takie, w którym gracz przejmuje w jednej turze liczbę kamieni zapewniającą mu zwycięstwo. To tak, jakby grając w

²Szczegółowe zasady standardowej odmiany gry *Teeko* zamieszczone w dodatku B.24.

¹Szczegółowe zasady gry *Dakon* zamieszczone w dodatku B.4.

¹W Dakon-ie rozsiewanie odbywa się w kierunku przeciwnym do ruchu wskazówek zegara.

RYSUNEK 2.13. Plansza do gry z rodziny *Mankala*RYSUNEK 2.14. Plansza do gry *Ohvalhu* (Dakon-8 prawoskrętny)

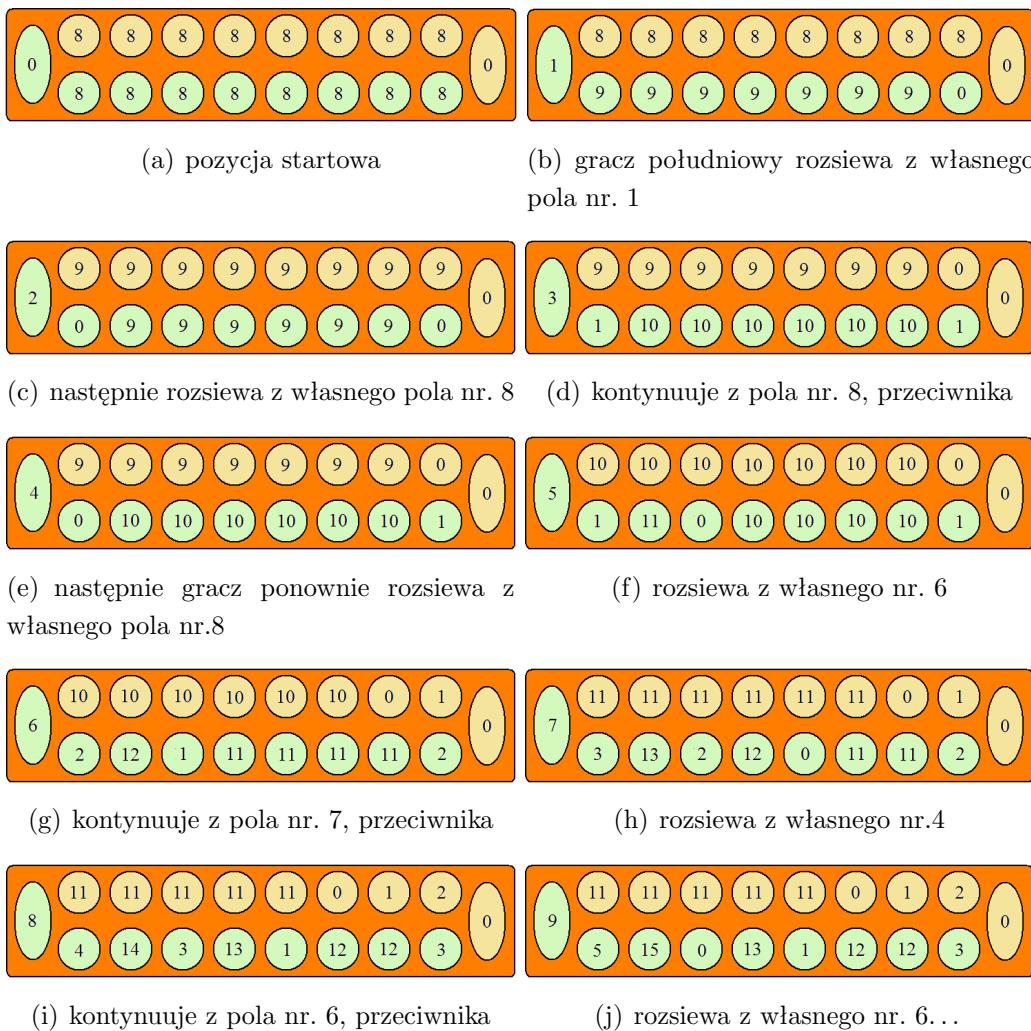
Bilard nie dopuścić przeciwnika do stołu, wbijając po kolej wszytkie swoje bile.

Planszę do tej wersji gry przedstawiono na rys. 2.14. Pola startowe ponumerowane są od 1 do 8, od prawej do lewej strony patrząc z pozycji gracza. Na rys. 2.15(a) - 2.15(c) przedstawiono początkowe dziewięć rozsiań otwarcia wygrywającego w *Ohvalhu*. W tym wariantie gry na każdego z graczy przypada osiem pól. Poniżej przedstawiono wszystkie rozsiania, należące do tego otwarcia:

1, 8, 8, 6, 4, 6, 2, 3, 7, 4, 5, 1, 8, 8, 3, 8, 4, 8, 7, 8, 5, 2, 7, 8, 6, 5, 6, 3, 7, 4, 5,
2, 5, 8, 8, 6, 8, 3, 8, 5, 8, 7, 4, 8, 7, 8, 7, 8, 8, 6, 8, 7, 8, 4, 8, 6, 8, 3, 8, 6, 8, 5,
8, 6, 1, 8, 7, 8, 5, 8, 4, 6, 7, 8, 8, 5, 6, 8, 3, 8, 1, 8, 7, 8, 2, 8, 6, 8, 5, 8, 6, 8, 3

Każda liczba określa pole startowe, z którego rozsiewane są ziarna. Podane są tylko rozsiania od strony gracza. Jeśli któryś z rozsiań kończy się na dowolnym polu przeciwnika, to jest ono kontynuowane poczynając od tego pola. Odmienną czcionką zaznaczono rozsiania przedstawione na wcześniejszych rysunkach. Imponujący jest fakt, że do znalezienia tego rozwiązania, nie były użyte żadne maszyny liczące.

Jeroen Donkers, Alex de Voogt i Jos Uiterwijk [16] w 2000 roku, używając Standardowego PC z procesorem Pentium II 300 MHz oraz algorytmu przeszukiwania z nawracaniem napisanego w języku Java, znaleźli otwarcia wygrywające aż do wersji *Dakon-18*, a także ich charakterystyczne rodzaje, takie jak:



RYSUNEK 2.15. Dziesięć pierwszych stanów gry otwarcia wygrywającego w *Ohvalhu* (cyfra na danym polu oznacza liczbę znajdujących się w nim ziaren)

- otwarcie wygrywające przechwytyjące maksymalną liczbę kamieni przeciwnika;
- najkrótsze otwarcie wygrywające (najmniejsza liczba rozsiań);
- otwarcie wygrywające zawierające najmniejszą liczbę rund¹.

Kolejną grą w rodzinie *Mankala*, do której rozwiązywania już musiano wspomóc się komputerem to *Kalah(m,n)*¹ [25]. Jest to komercyjny wariant gry *Mankala*, szczególnie popularny z USA. Dla małych instancji gry zostały skonstruowane bazy

¹W Dakonie każda runda składa się z pewnej liczby rozsiań. Dla typowego gracza łatwiejsze do zapamiętania są otwarcia, składające się z jak najmniejszej liczby rund.

¹Szczegółowe zasady gry *Kalah* zamieszczono w dodatku B.13.

danych zawierające wszystkie ich stany (rozwiązań silne). Większe instancje gry, ($4 \leq m \leq 6, 1 \leq n \leq 6$) z wyjątkiem (6,6) zostały lekko rozwiązane przy użyciu algorytmu „ $MTD(f)$ z iteracyjnym pogłębianiem” oraz zastosowaniem wielu optymalizacji takich jak: „sortowanie ruchów”, „tablice transpozycji”, „futility pruning” oraz „bazę końcówek”. Program przeszukujący napisany był w języku C++. Wartości tych obliczeń, z pozycji gracza rozpoczynającego zostały przedstawione w tab. 2.2.

TABLICA 2.2. Wyliczone wartości optymalnej rozgrywki gry Kalach(m,n)

m/n	1	2	3	4	5	6
1	remis	przegrana	zwycięstwo	przegrana	zwycięstwo	remis
2	zwycięstwo	przegrana	przegrana	przegrana	zwycięstwo	zwycięstwo
3	remis	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo	przegrana
4	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo	remis
5	remis	remis	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo
6	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo	zwycięstwo	nie znana

Kolejną rozwiązaną grą była *Renju*². Jest to stara gra planszowa, pochodząca z Japonii. Powstała poprzez wprowadzenie wielu dodatkowych, asymetrycznych zasad do swojej poprzedniczki – *Go-Moku*. Wprowadzenie tych różnic miało na celu zmniejszenie przewagi gracza czarnymi. Jednak w 2001 roku János Wágner i István Virág wykazali, że gracz ten nadal posiada strategię wygrywającą.

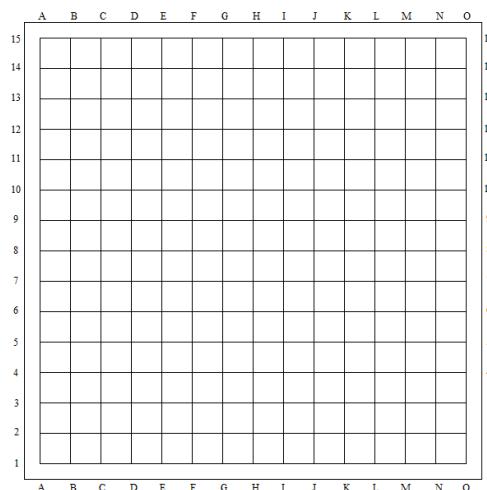
TABLICA 2.3. Pojedynek w *Renju*, gracze używają optymalnych strategii

1. h8	2. h7	3. i7	4. g9	5. j6	6. i8	7. i6	8. g6	9. j9	10. k5
11. k6	12. l6	13. j7	14. j8	15. k8	16. l9	17. l7	18. m6	19. j4	20. i4
21. k7	22. m7	23. i5	24. h4	25. g7	26. h6	27. h5	28. g4	29. f5	30. g5
31. j10	32. f4	33. e4	34. e3	35. d2	36. g3	37. g2	38. n8	39. o9	40. i9
41. i10	42. h11	43. k10	44. k9	45. h10	46. g10	47. l10			

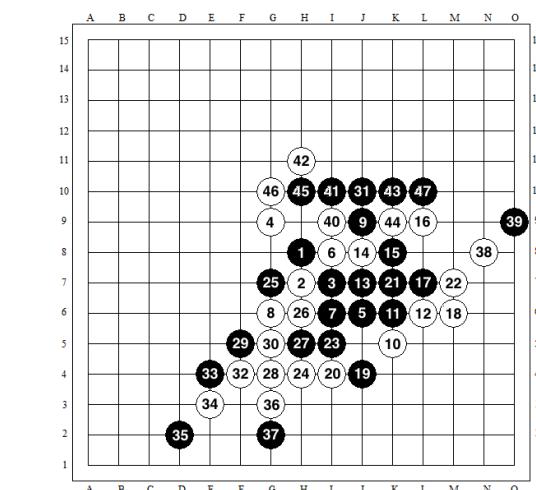
W tab. 2.3 przedstawiono przebieg najkrótszej rozgrywki znalezionej przez wspomnianych autorów. Prowadzi ona do wygranej czarnego przy najskuteczniejszej obronie przeciwnika. Nie jest to jednak najkrótsza możliwa rozgrywka, gdyż znalezienie dowolnej wygrywającej strategii czarnego, rozwiązuje grę, a właśnie to było nadziednym celem tych badań. Przeanalizujmy pierwsze pięć posunięć, w których to można

²Szczegółowe zasady gry *Renju* zamieszczone w dodatku B.23.

zauważyc wprowadzone restrykcje. W pierwszym posunięciu grający tymczasowo czarnymi musi postawić swój pionek na środku planszy (pole **h8**). Jest to standardowa zasada obowiązująca w *Go-moku*. W kolejnym kroku przeciwnik, grający tymczasowo białymi, musi położyć pionek na jednym z ośmiu pól, sąsiadujących z czarnym pionkiem. Wybiera więc pole **h7**. Teraz ruch grającego tymczasowo czarnymi jest ograniczony, przez zasady gry, do środkowego kwadratu o boku 5x5 pól. Czarny stawia więc swój pionek na **i7**. Biały może teraz zdecydować się na zmianę koloru, powinien to zrobić, ponieważ grający teraz czarnymi posiada strategię wygrywającą. Kolejny ruch należy do grającego teraz białymi i nie jest on w żaden sposób ograniczony. Następnie czarny proponuje dwa, asymetryczne posunięcia swoich pionków, a biały wybiera jedno z nich, mianowicie **g9**. Kolejny ruch należy do białych, które kładą swój pion na **j6** i teraz rozgrywka toczy się już „normalnym” biegiem. Należy jednak pamiętać, że pewne ograniczenia cały czas dotyczą posunięć gracza czarnego. Zostały one dokładnie opisane w zasadach gry. Na rys. 2.16(b) przedstawiono planszę z uwidocznionym przebiegiem tego pojedynku.



(a) plansza do gry

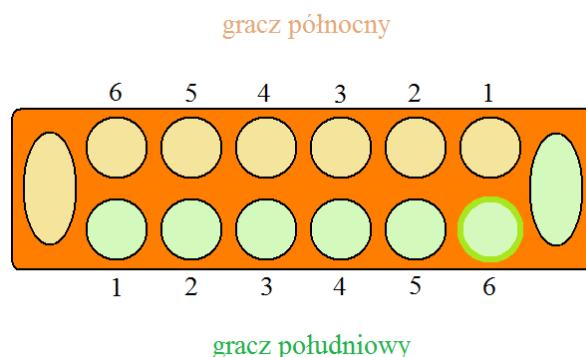


(b) najlepsza obrona białych, czarne wygrywają w 47 posunięciach

RYSUNEK 2.16. Gra *Renju*

Autorzy tego rozwiązania użyli programu napisanego w języku Pascal, który bazował na algorytmie przeszukiwania „Threat-sequence”, zaproponowanym przez Allisa i innych podczas pracy nad *Go-Moku*. Rozwiązywanie gry pochłonęło 9000 godzin pracy komputera PC z procesorem Pentium 200 MHz oraz bazą danych o rozmiarze 7 MB.

Jak opisano wcześniej w 1990 roku przewidywano, że do 2000 roku powstanie program rozwiążający bardzo popularny wariant gry *Mankala – Awari*¹. Gra pochodzi z Afryki, a jej wiek ocenia się na 3500 lat. Pomimo to nadal jest praktykowana na całym świecie. Gra, przy optymalnej rozgrywce przez obydwu graczy, prowadzi do remisu. Na rys. 2.17 przedstawiono planszę do gry z zaznaczonym (zielonym), jedynym remisującym polem. Rozpoczęcie gry przez gracza południowego, z któregokolwiek z pozostałych pól, prowadzi przy optymalnych posunięciach przeciwnika do jego przegranej.

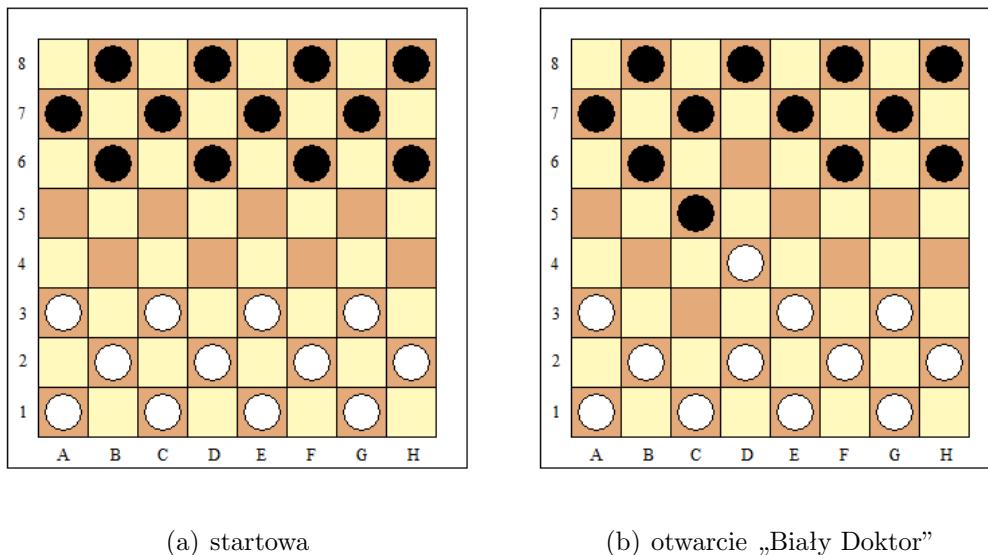


RYSUNEK 2.17. Plansza do gier *Mankala*: *Awari*, *Dakon-6* i *Kalah*(6,4)

Dopiero w 2002 roku moce obliczeniowe komputerów i pojemności baz danych były wystarczające aby zmierzyć się z tym wyzwaniem. Henri Bal i John Romein silnie rozwiązali tę grę przy użyciu równoległej jednostki obliczeniowej, składającej się z 72 dwuprocessorowych jednostek produkcji IBM, połączonych szybką siecią Myrinet 2Gb/s (full-duplex). Łącznie wykorzystali 144 procesory Pentium III 1.0 GHz, 72 GB pamięci operacyjnej oraz przestrzeń dyskową o powierzchni 1,4 TB. Do konstrukcji bazy danych użyto nowego, równoległego algorytmu, bazującego na analizie wstecznej. Rozwiążanie zajęło zaskakująco mało czasu pracy komputera, bo jedynie 51 godzin.

¹Szczegółowe zasady gry *Awari* zamieszczone w dodatku B.2.

W 2007 roku Jonathan Schaeffer rozwiązał *Warcaby angielskie*¹ [50, 51, 53]. Którykolwiek z graczy grając optymalnie nigdy nie przegra. Już w 1994 roku twórcy programu „Chinook”, mistrza świata w tej kategorii oświadczyli, że otwarcie zwane „Białym Doktorem”, zamieszczone na rys. 2.18(b), zapewnia białym remis. Po trzech latach udowodniono, że żadne z pozostałych otwarć nie zapewnia wygranej białym.



RYSUNEK 2.18. Pozycje w *Warcabach angielskich*

Ze względu na zarówno wysoką złożoność przestrzeni jak i wysoki współczynnik rozgałęzienia, procedura dowodząca składała się z czterech części:

Baza danych końcówek (przeszukiwanie wstecz) – baza danych $4 * 10^{13}$ pozycji, zawierających nie więcej niż dziesięć bierek na planszy. Wszystkie te pozycje zostały silnie rozwiązane i skompresowane do 237 GB – 154 pozycje/Bajt. Użyto własnego algorytmu kompresji pozwalającego na szybką lokalizację i dekompresję danych [52]. Pozwala to na natychmiastowe określenie wyniku pozycji i uniknięcie wielokrotnej analizy tych samych stanów gry. Przykładowo, już dla bazy zawierającej siedem pozycji najdłuższa wygrana rozgrywka trwa 253 posunięcia [59].

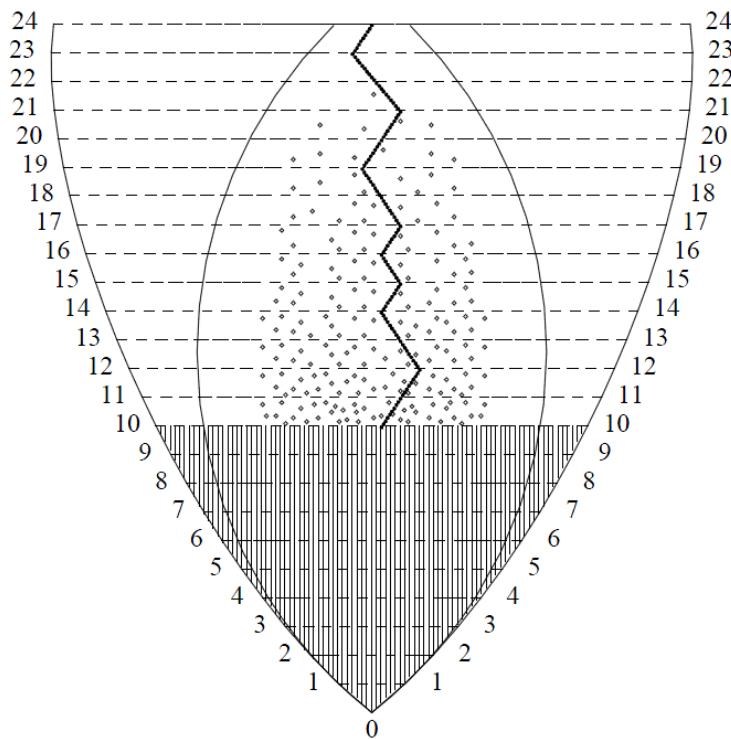
Menedżer drzewa dowodzącego (zarządzanie przeszukiwaniem) – komponent ten jest odpowiedzialny za utrzymanie oryginału drzewa dowodzącego. Za pomocą algorytmu „Proof-number search” oraz wartości wyliczonych przez

¹Szczegółowe zasady gry *Warcabów angielskich* zamieszczone w dodatku B.26.

„analizator dowodzący” wyszukuje najlepsze wierzchołki do rozwiązania, które następnie magazynuje.

Analizator dowodzący (przeszukiwanie wprzód) – komponent, który otrzymuje pozycję do rozwiązania i za pomocą algorytmów „17÷23-ply $\alpha\text{-}\beta$ ”¹ i „Df-pn”² określa ich wartość. Wartość ta może być rozwiązana: {wygrana, przegrana, remis}; częściowo rozwiązana: {co najmniej remis, co najwyżej remis} lub heurystycznie określona za pomocą liczby.

Debiut (doświadczenie eksperckie) – pojedyncza linia „najlepszej rozgrywki” zidentyfikowana na podstawie literatury warcabowej. Jest przewodnikiem dla „menedżera drzewa dowodzącego” jako inicjalna linia do poszukiwań. Menedżer przekazuje do rozwiązania analizatorowi ostatnią pozycję z tej linii, zapamiętuje wynik, a następnie cofa się o jedno posunięcie wzdłuż tej linii, w ten sposób kontynuując obliczenia, aż osiągnie korzeń drzewa.



RYSUNEK 2.19. Przestrzeń stanów w *Warcabach angielskich* [51]

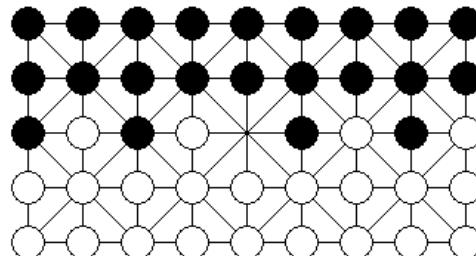
¹Wariant przeszukujący drzewo gry na głębokość 17 – 23 posunięć.

²Wariant algorytmu „Proof-number search”.

Na rys. 2.19 przedstawiono rozmieszczenie poszczególnych elementów procedury dowodzącej w przeszukiwanej przestrzeni stanów, układającej się w kształt kieliszka. Liczby w pionie określają ile pozostało bierek na planszy. Szerokość kieliszka określa (w skali logarytmicznej) wielkość podprzestrzeni związanej z określoną liczbą bierek. Część dolna, wypełniona winem, to „baza danych końcówek”. Pojedyncze bąbelki powyżej bazy danych to wierzchołki magazynowane przez „menedżera drzewa dowodzącego”, a powyginana słomka to właśnie „doświadczenie eksperckie”.

Cały dowód w postaci kompletnej strategii obydwu graczy jest dostępny na stronie programu Chinook [9]. Zmagazynowane drzewo składa się z 10^7 pozycji. Całkowita liczba wierzchołków przeszukanych w dowodzie $\approx 10^{14}$. Najdłuższa analizowana ścieżka ma głębokość 154 posunięć. Liczba wierzchołków w minimalnym drzewie dowodzącym wynosi 223.178 i posiada głębokość 46 wierzchołków. Średnia liczba użytych procesorów do obliczeń to siedem z 1,5 do 4 GB pamięci operacyjnej.

W tym samym roku Maarten Schadd i inni rozwiązali narodową grę Madagaskaru – *Fanorona*¹ [49]. Okazało się, że analogicznie jak w *Warcabach angielskich* optymalna rozgrywka obydwu graczy zawsze kończy się remisem.

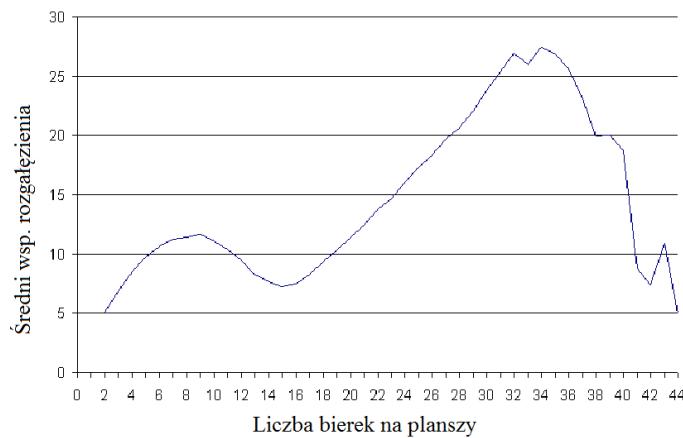


RYSUNEK 2.20. Pozycja początkowa w *Fanorонie*

W tym przypadku wykorzystano dwie metody: „analizę przeszukiwania wstecz” i algorytm „Proof-number search”. Ponieważ *Farona* jest, podobnie jak wcześniej wspomniane *Warcaby*, grą zbieżną i tutaj można było zastosować bazę końcówek szczególnie, że specyfika tej gry polega na tym, iż większa część rozgrywki składa się z pozycji zawierających niewielką liczbę pionków, zawsze po 21 ruchach na planszy zostaje ich najwyżej 8. Lokalne maksimum funkcji średniego współczynnika rozgałęzienia, przedstawionej na rys. 2.20, nie przekracza 12. Również końówka gry nie jest trywialna, co uzasadnia sens zastosowania bazy danych.

Ponieważ liczba wszystkich możliwych pozycji gry jest olbrzymia skonstruowanie

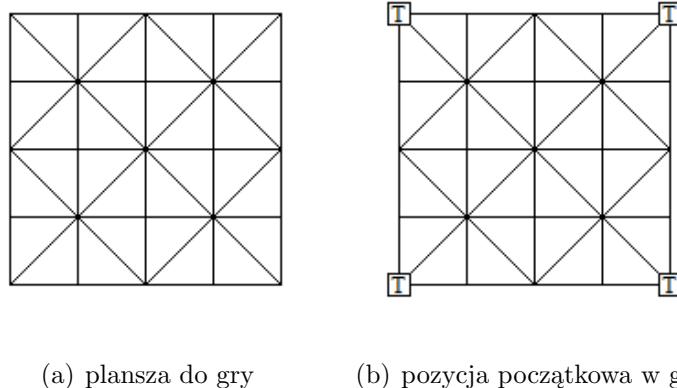
¹Szczegółowe zasady gry *Fanorona* zamieszczono w dodatku B.6.



RYSUNEK 2.21. Średni współczynnik rozgałęzienia jako funkcja liczby pionków na planszy [49] występujący w grze *Farona*

bazy danych do pozycji inicjalnej okazało się niemożliwe. Dlatego zastosowano typowe przeszukiwanie wprzód z użyciem algorytmu „Proof-number search”. Rozwiążanie gry zajęło miesiąc czasu obliczeń komputera PC z procesorem pentium IV 3.0 GHz i 256 MB pamięci operacyjnej.

Kolejną grą rozwiązaną w 2007 roku jest starożytna gra z Nepalu, znana pod takimi nazwami jak: *Bagh-Chal* lub *Tygrysy i Kozy*¹.

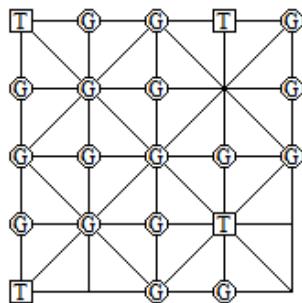


RYSUNEK 2.22. Gra *Tygrysy i Kozy*

Yew Jin Lim udowodnił, że optymalna rozgrywka prowadzi do remisu. Przestrzeń stanów gry została podzielona na pięć części, zależnie od liczby zbitych kóz. Pozwoliło to na rozwiązywanie gry etapami. Udowodnienie, że tygrys może zabić jedną

¹Szczegółowe zasady gry *Tygrysy i Kozy* zamieszczone w dodatku B.25.

lub dwie kozy zajęły dzień obliczeń komputerowi PC z procesorem pentium 4 przy zastosowaniu standardowego „przeszukiwania z przycinaniem α - β ” i „tablicami haszującymi”. Dla trzech kóz ten sam komputer pracował przez miesiąc bez rezultatu. Dopiero w przybliżeniu tygodni pracy algorytmu „równoległego przeszukiwania α - β ” z rozłącznymi poddrzewami dla każdego procesu zarządzającego własną „tablicą haszującą” (zawierającą 8 milionów wpisów) i redukcją „pozycji symetrycznych”, uruchomionego na klastrze zbudowanym z 48 procesorów¹ Dual Pentium 3, pozwoliło udowodnić, że tygrysy mogą zabić trzy kozy.



RYSUNEK 2.23. Ruch tygrysów, jednak kozy wygrywają w 5 posunięciach

Z braku wiedzy eksperckiej na temat gry autorzy zdecydowali się na zastosowanie systemu ko-ewolucyjnego z dwiema populacjami. Populacja składała się z 20 sieci neuronowych, każda jako funkcja ewolucyjna z losowo dobranymi wagami. Podobne systemy zostały już wcześniej sprawdzone, osiągając poziom ekspercki w warcabach (2001) [18] oraz Kalach (2003) [36]. Podczas jednego etapu ewolucji każda sieć neuronowa konkurowała z czterema losowo wybranymi sieciami, a następnie dziesięć najlepszych wraz z potomkiem utworzonym poprzez mutację swoich wag, przechodziło do kolejnej generacji. Uzyskane w ten sposób (po tysiącu pokoleń) sieci neuronowe, zostały zastosowane podobnie jak w przypadku *Warcabów* jako wiedza ekspercka i pozwoliły na naprowadzenie algorytmów przeszukujących. Pozwoliło to udowodnić, że rozgrywka w *Tygrysy i Kozy* prowadzi do remisu. Obliczenia te wymagały dwóch miesięcy pracy 10 procesorów Pentium 4 Xeon.

¹W praktyce oprogramowanie pozwalało na wykorzystanie tylko ośmiu procesorów.

Przedstawiając historię rozwiązywania gier autor starał się przedstawić rozwiązania najważniejszych i najbardziej znanych gier planszowych dla dwóch osób, deterministycznych, z pełną informacją. Gry trywialne jak *Kółko i Krzyżyk*, bądź określone wersje znanych gier takich jak *Hex* na mniejszej planszy specjalnie zostały pominięte. Poniżej w tab. 2.4 przedstawiono jeszcze raz wszystkie te rozwiązania w porządku chronologicznym. Jednak należy mieć na uwadze, że ograniczenia czasowe oraz przestrzenne tej pracy mogą być przyczyną pominięcia twórców pewnych rozwiązań, które z uzasadnionych przyczyn powinny się znaleźć w tym opracowaniu.

TABLICA 2.4. Lista rozwiązanych gier logicznych dwuosobowych z pełną informacją

<i>Id.</i>	<i>Rok</i>	<i>Gra</i>	<i>Autor</i>	<i>zl. st.¹</i>	<i>zl. dr.²</i>	<i>ref.³</i>	<i>rozw.⁴</i>
1.	1949	Hex	John Nash	10^{57}	10^{98}	[7,8]	ultra-lekkie
2.	1980	Qubic	Oren Patashnik	10^{30}	10^{34}	[39]	lekkie
3.	1988	Czwórki	James D. Allen oraz Victor Allis	10^{14}	10^{21}	[1,2]	lekkie
4.	1990	Qubic	Victor Allis i Patrick Schoo	10^{30}	10^{34}	[3]	silne
5.	1992	Go-Moku	Victor Allis	10^{105}	10^{70}	[3–5]	lekkie
6.	1993	Młynek	Ralph Gasser	10^{10}	10^{50}	[19]	lekkie
7.	1995	Czwórki	John Tromp	10^{14}	10^{21}	[60]	silne
8.	1996	Pentomino	Hilarie K. Orman	10^{12}	10^{18}	[37]	lekkie
9.	1998	Quarto Twist	Luc Goossens	b.d.	b.d.	–	lekkie
10.	1998	Teeko	Guy Steele	b.d.	b.d.	[61]	lekkie
11.	2000	Dakon-6⁵	Jeroen Donkers i inni	10^{15}	10^{33}	[16]	lekkie
12.	2000	Kalah(6,4)	Geoffrey Irving i inni	10^{13}	10^{18}	[25]	lekkie
13.	2001	Renju	János Wágner i István Virág	10^{105}	10^{70}	[63]	lekkie
14.	2002	Avari	Henri Bal i John Romein	10^{12}	10^{32}	[45]	silne
15.	2003	Quarto	Arthur Holshouser i Harold Reiter	b.d.	b.d.	[24]	lekkie
16.	2007	Warcaby angielskie	Jonathan Schaeffer	10^{21}	10^{31}	[9,50,51,53]	lekkie
17.	2007	Fanorona	Maarten Schadd	10^{22}	10^{46}	[49]	lekkie
18.	2007	Tygrysy i kozy	Yew Jin Lim	10^{10}	10^{31}	[34]	lekkie

¹złożoność przestrzeni stanów gry, **b.d.** – brak danych.²złożoność drzewa gry, **b.d.** – brak danych.³odnośniki do bibliografii.⁴rodzaj rozwiązania (ultra-lekkie, lekkie, silne).⁵wcześniej łatwo rozwiązana przez graczy z Malediw.

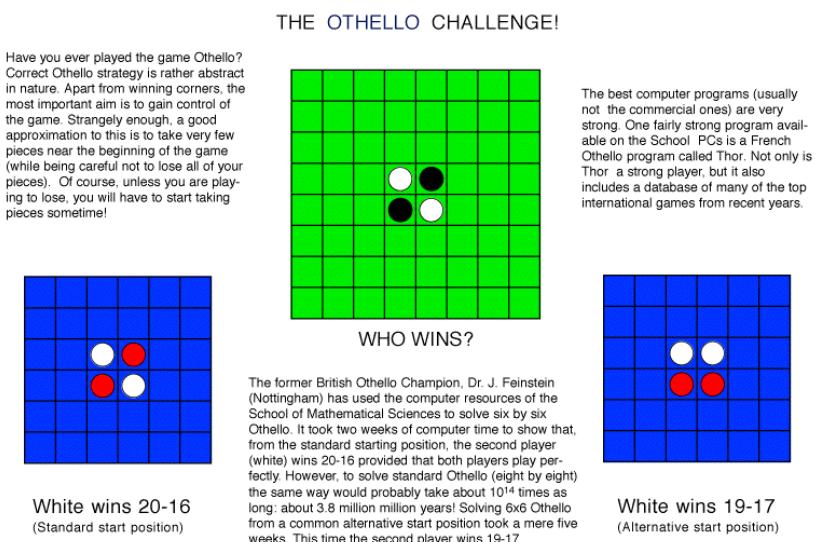
2.4. Perspektywy osiągnięć w rozwiązywaniu gier

Analogiczne do przewidywań z 1990 roku postawiono hipotezy dla programów, które będą grały w 2010 roku [23]. Prognozy te są zamieszczone w tab. 2.5.

TABLICA 2.5. Hipotezy z 2000 na 2010 rok dotyczące siły programów grających w gry z olimpiad komputerowych

<i>rozwiązywany</i>	<i>ponad mistrzowski</i>	<i>mistrz świata</i>	<i>mistrz</i>	<i>amator</i>
Awari	Szachy	Go (9×9)	Brydż	Go
Othello	Warcaby polskie	Szachy chińskie	Shogi	
Warcaby angielskie	Scrabble	Hex		
	Backgammon	Amazons		
	Lines of Action			

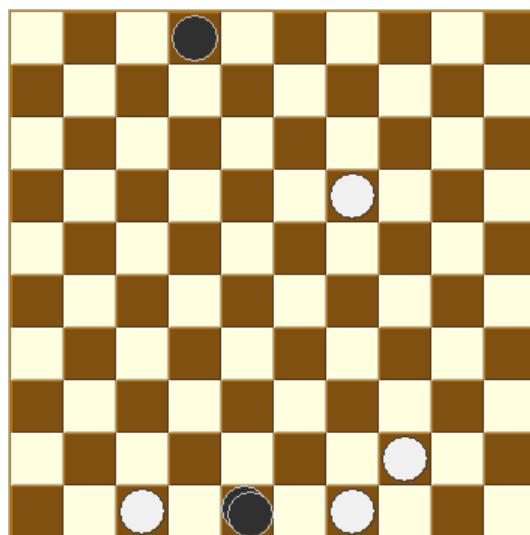
Najbardziej prawdopodobnym wydaje się rozwiązywanie w najbliższym czasie gry *Othello*. Jest to jedyna gra, co do której wspomniane przewidywania jeszcze się nie potwierdziły. Co prawda Joel Feinstein już w 1993 roku, wykorzystując przeszukiwanie α - β , po dwóch tygodniach obliczeń wykazał, że rozgrywka w *Othello* na planszy o wymiarach (6×6) jest wygrana dla gracza drugiego, jednak szacuje się, że rozwiązanie klasycznej wersji gry tą samą metodą zajęłoby, aż $3,8 \times 10^{12}$ lat.



RYSUNEK 2.24. Poster ze strony domowej Joela Feinstein'a

Na rys. 2.24 przedstawiono poster ze strony Joela Feinstein'a informujący, że rozgrywka w *Othello* (6×6) przy standardowej pozycji startowej, kończy się dla rozpoczynającego w najlepszym wypadku jego przegrana z wynikiem 16 – 20. Alternatywna pozycja startowa, jest dla niego bardziej optymalna, ponieważ może on przegrać jedynie dwoma punktami, z wynikiem 17 – 19.

Po rozwiązaniu *Warcabów angielskich* przyszła kolej na *Warcaby polskie*. Jest to dużo większe wyzwanie. Nie tylko z powodu większych wymiarów warcabnicy (10×10), ale przede wszystkim z większej „ruchliwości” bierek. Szczególna różnica jest widoczna w możliwościach damek. Mogą one przemieszczać się o dowolną liczbę pól i zatrzymywać się na dowolnym, pustym polu za przeciwnikiem. Powoduje to, w połączeniu z możliwością kontynuacji bicia, znaczne zwiększenie współczynnika rozgałęzienia drzewa gry, szczególnie w jej końcówce.

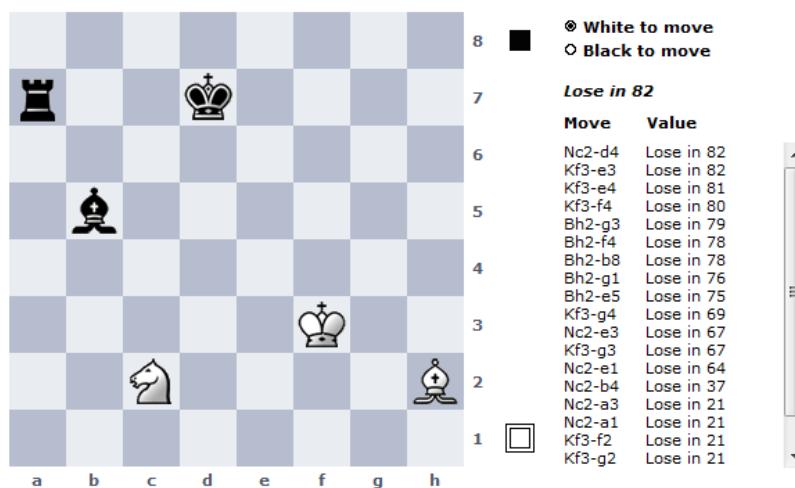


RYSUNEK 2.25. Pozycja początkowa najdłuższej dotychczas znalezionej końcówki w *Warcabach polskich* [20]

Dotychczas udało się opracować bazę końcówek, co najwyżej dla kombinacji pięciu pionków i dwóch damek oraz sześciu i mniejszej liczby dowolnych bierek na planszy. Najdłuższa dotąd znaleziona końcówka w *Warcabach polskich* składa się z 129 posunięć, a jej pozycję startową przedstawiono na rys. 2.25. Pomimo iż czarne są już w posiadaniu damki, przewaga liczenna oraz pozycja białych pionów zapewnia im zwycięstwo.

Trudno nie wspomnieć o grze w *Szachy*, która co prawda nie doczekała się jeszcze rozwiązania i nie wydaje się aby szybko to miało nastąpić, jednak bazy wszystkich

możliwych końcówek dla liczby bierek nie większej niż sześć są na dzień dzisiejszy opracowane i powszechnie dostępne w internecie. Wielkość takiej bazy danych wynosi 1146 GB. Przykładową końcówkę przedstawiono na rys. 2.26. Biale zaczynają i najpóźniej po 82 ruchach przegrywają grę.



RYSUNEK 2.26. Przykładowa końcówka w *Szachach*

O złożoności tej gry może świadczyć fakt, że największa plansza dla której znane są wartości wszystkich pozycji to plansza o wymiarach (3×3) . Najdłuższa rozgrywka na takiej szachownicy może toczyć się przez 15 rund [30], a pozycję startową takiej partii pokazano na rys. 2.27. Wygrana i pierwszy ruch należą do białych.



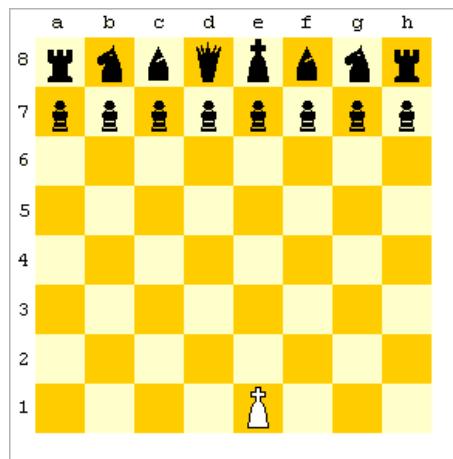
RYSUNEK 2.27. Pozycja początkowa najdłuższej rozgrywki w *Szachach* (3×3) [31]

Spektakularne zwycięstwo Deep-Blue [13] nad arcymistrzem szachowym – Garrym Kasparov’em potwierdza, zgodnie z przewidywaniami, ponad mistrzowski poziom programu grającego w *Szachy*.

Ciekawostką jest fakt, że wariant szachów o nazwie *Maharadża i Sepoje*¹ został rozwiązywany. Cytując Hansa Bodlaendera [10] „Ostrożna gra czarnego powinna mu

¹Szczegółowe zasady gry *Maharadża i Sepoje* zamieszczone w dodatku B.15.

zapewnić zwycięstwo, co jednak nie jest łatwe, ponieważ w wielu przypadkach, Maharadża, przebijając się przez linię czarnych, ma duże szanse na wygraną". Planszę do gry przedstawiono na rys. 2.28.



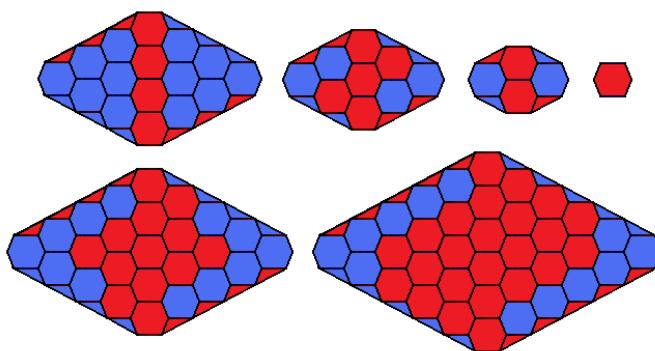
RYSUNEK 2.28. Pozycja początkowa w grze *Maharadža i Sepoje* [10]

Gra *Hex*, jako pierwsza gra z ultra-lekkim rozwiązyaniem, dotychczas doczekała się jedynie słabego rozwiązywania na planszy o maksymalnych wymiarach (9×9) oraz silnego rozwiązywania na planszy o wymiarach (8×8). Poniżej przedstawiam w kolejności chronologicznej osiągnięcia związane z rozwiązywaniem tej gry:

- W 1990 roku Bert Enderton [17] znalazł strategię wygrywającą gracza rozpoczynającą dla *Hex'a* granego na planszy o wymiarach do (6×6) włącznie.
- W 1999 roku Jack van Rijswijck [43] napisał program QueenBee [42] silnie rozwiązuje tą grę na planszy o wymiarach do (6×6) włącznie.

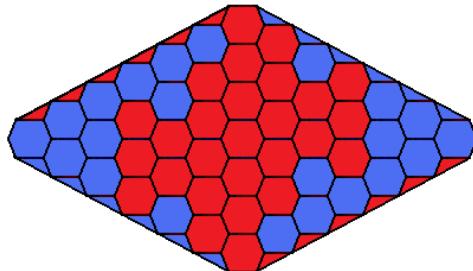
Wyniki tych rozwiązań pokazano na rys. 2.29. Wygrywa gracz, w którego kolorze jest dane pole zakładając, że na tym polu rozpoczyna rozgrywkę grający kolorem czerwonym.

- W latach kolejno 2001, 2002 i 2003 Jin Yang i inni podawali kolejne strategie wygrywające dla gry o wymiarach odpowiednio: (7×7) [67], (8×8) [68] i (9×9).
- W 2003 roku Hayward i inni [21] silnie rozwiązali *Hex'a* na planszy o wymiarach (7×7). Wykorzystali własnego pomysłu algorytm przeszukujący z nawrotami oraz silnikiem wirtualnych połączeń (VCE). Do obliczeń posłużył im



RYSUNEK 2.29. Rozwiązywanie ruchów otwierających w *Hex'ie* o rozmiarach do 6×6 pól włącznie (kolor przedstawia zwycięzcę, jeśli czerwony rozpoczyna na danym polu)

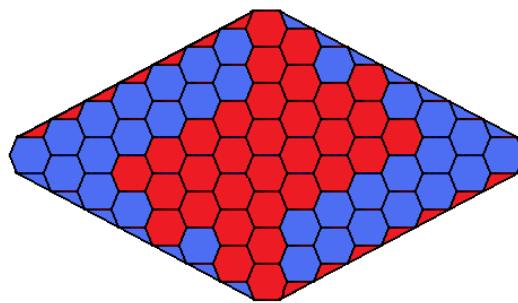
Komputer PC z procesorem AMD Athlon 1800+ i 512 MB pamięci operacyjnej, któremu rozwiązywanie wszystkich 49 pozycji startowych zajęło 615 godzin. Na rys. 2.30 podano wyniki optymalnych rozgrywek. Wygrywa gracz, w którego kolorze jest dane pole, zakładając że na tym polu rozpoczyna rozgrywkę grający kolorem czerwonym.



RYSUNEK 2.30. Rozwiązywanie ruchów otwierających w *Hex'ie* (7×7) (kolor przedstawia zwycięzcę, jeśli czerwony rozpoczyna na danym polu)

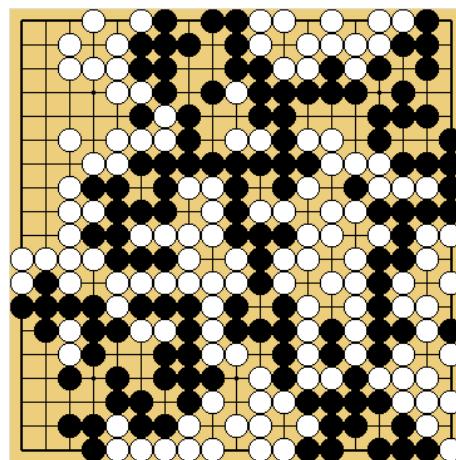
- W 2009 roku Philip Henderson i inni [22], bazując na algorytmie wymyślonym przez Haywarda [21], podali silne rozwiązanie tej gry na planszy o wymiarach (8×8) . To rozwiązanie zostało zamieszczone na rys. 2.31. Oprócz wspomnianego algorytmu zastosowali własne nowatorskie rozwiązania, takie jak: „graph theoretic inferior cell methods”, „combinatorial decompositions”, „proof set shrinking” i „transposition deductions”. Obliczenia wymagały 301 godzin pracy komputera PC z procesorem Intel Core 2 Quad i 2 GB pamięci operacyjnej.

Największą tajemnicą owianą jest gra *Go*. Widok planszy z przykładowym stanem



RYSUNEK 2.31. Rozwiążane ruchy otwierające w *Hex'ie* (8×8) (kolor przedstawia zwycięstwę, jeśli czerwony rozpoczyna na danym polu)

końcowej fazy tej gry przedstawiono na rys.2.32. Została rozwiązana w 2003 roku przez Erika van der Werf'a [62] dla planszy o wymiarach zaledwie 5×5 , czyli o ponad 14-krotnie mniejszej liczbie pól niż ma to miejsce w oryginalne.

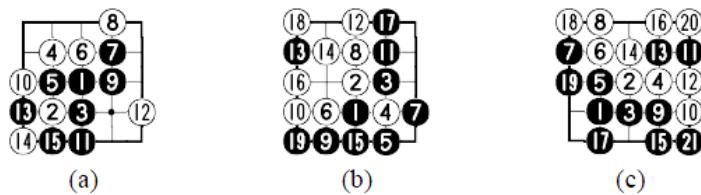


RYSUNEK 2.32. Faza końcowa gry *Go*

Do obliczeń posłużyono się komputerem PC z procesorem Pentium IV 2.0 GHz z algorytmem przeszukiwania α - β , rozszerzonym o tablice transpozycji, uwzględnienie symetrii, wewnętrzne bezwarunkowe ograniczenia oraz zaawansowane sortowanie ruchów. Pozwoliło to udowodnić, że *Go* na planszy (5×5) jest wygrywające dla gracza rozpoczynającego¹.

Na rys. 2.33 przedstawiono trzy przykłady optymalnej rozgrywki w *Go*, najlepsze pierwsze posunięcie dla gracza rozpoczynającego to zajęcie centralnego pola planszy. Największą chwałą tej gry jest fakt, że najlepsze programy nie wzbiły się ponad amatorski poziom. Człowiek, pomimo że został zepchnięty z podium w przypadku

¹W *Go* rozpoczyna gracz grający czarnymi pionami.



RYSUNEK 2.33. Trzy przykłady optymalnej rozgrywki w *Go* (5×5) [62]

takich gier jak *Szachy* bądź *Warcaby*, tutaj nieprzerwanie jest mistrzem, manifestując przewagę umysłu ludzkiego nad maszyną.

Wiemy, że komputery pozwalają nam odpowiedzieć na pytanie: jakiego wyniku możemy się spodziewać grając optymalnie? Pozwalają również pokazać nam optymalną strategię gry. Jednak otwarte pozostają następujące trzy pytania: [15]

- czy wiedzę zgromadzoną przez komputery można przetworzyć w jakiś sposób na proste reguły, które mogłyby być użyteczne dla człowieka?
- czy byłyby to ogólne zasady, czy mnóstwo doraźnych przepisów?
- czy metody mogą być przekazywane pomiędzy grami?, czy istnieją ogólne metody dla pewnych kategorii gier, czy każda gra jest odrębna kategorią rządzającą się swoimi prawami?

Rozdział 3

Metody rozwiązywania gier logicznych

Podczas licznych zmagań z pisaniem programów grających w gry logiczne wymyślono wiele metod i algorytmów. Powstały algorytmy ogólne, które można zastosować do prawie każdego problemu oraz specyficzne, mające bardzo wąskie zastosowanie. Można je podzielić na: algorytmy deterministyczne, wyznaczające dokładną wartość pozycji w grze oraz algorytmy heurystyczne, które ze względu na olbrzymią przestrzeń drzewa gry posługują się funkcją oceny pozycji. Funkcja taka musi być ściśle powiązana z wiedzą ekspercką na temat gry. Wiedza ta może być zdobywana przez pokolenia graczy, jak to ma miejsce w znanych grach: *Warcaby*, *Szachy* lub znaleziona za pomocą algorytmów ewolucyjnych, przykładowo: *Backgammon* lub *Tygrysy i Kozy*.

W niniejszej pracy skupiono się nad algorytmami spełniającymi dwa kryteria:

- **ścisłość rozwiązania** – algorytmy mające za zadanie dokładne wyznaczanie wartości pozycji w grze;
- **uniwersalność** – algorytmy mające szerokie zastosowanie, które nie są zależne od specyfiki gry.

Wybór ten jest uzasadniony dwoma faktami. Pierwszym jest pojawienie się wielu prac związanych z zastosowaniem sztucznej inteligencji w grach i jeśli w ogóle opisują one algorytmy deterministyczne do rozwiązywania gier to przeważnie w marginalnym stopniu. Natomiast często przywoływane algorytmy bazujące na heurystycznej funkcji oceny, nie mogą zostać bezpośrednio użyte do rozwiązywania gry. Drugim jest olbrzymia liczba algorytmów szczegółowych, która jest porównywalna z liczbą swoistych problemów we wszystkich grach logicznych. Ponadto przeważnie są one ściśle powiązane ze specyfiką gry, co czyni je bezużytecznymi podczas zmagań z inną grą. Zrezygnowanie z ich prezentacji wydaje się naturalne, chociażby ze względu na ograniczenia czasowe i przestrzenne tej pracy.

3.1. Złożoność przestrzeni stanów i drzewa gry

3.1.1. Szacowanie rozmiaru przestrzeni stanów gry

Znajomość przestrzeni stanów jest jedną z możliwych ocen wielkości problemu jaki będzie stanowiło rozwiązywanie danej gry. Pomaga w konstruowaniu funkcji indeksującej opisanej w sekcji 3.1.4, oraz przy wyborze odpowiednich metod służących rozwiązywaniu gry. Dla małych przestrzeni stanów bardzo skuteczne okazują się metody siłowe (ang. *brute-force methods*) opisane w sekcji 3.2. Dla dużych należy szukać oparcia w metodach przeszukujących, bazujących na wiedzy (ang. *knowledge-based methods*), opisanych w sekcjach 3.3.1-23. Wyznaczając złożoność przestrzeni stanów dla gry planszowej można stosować różne, górne oszacowania. Które z nich będą możliwe do zastosowania oraz które będą bliższe rzeczywistej wartości szacowanej przestrzeni, zależy od charakteru gry.

Prostym i szybkim górnym oszacowaniem jest liczba wszystkich wariacji z powtórzeniami wyrazów ze zbioru dopuszczalnych stanów pola planszy o długości równej liczbie pól planszy. Pod pojęciem dopuszczalnych stanów pola planszy rozumie się puste pole gry bądź pole zajęte przez jedną (bądź więcej, jeśli reguły gry na to dopuszczają) figurę/bierkę. Przykładowo dla gry *Kółko i krzyżyk* jest to każda dziewięciowymiarowa wariacja zbioru trójelementowego. W skład tego trójelementowego zboru wchodzi: puste pole, pole zajęte przez kółko, pole zajęte przez krzyżyk. Liczba takich wariacji wyraża się wzorem:

$$\bar{V}_n^k = n^k \quad (3.1)$$

W przypadku omawianego *Kółka i krzyżyka* gdzie $n = 3$ a $k = 9$, wynosi ona:

$$\bar{V}_3^9 = 3^9 = 19.683 \quad (3.2)$$

Podobnie dla gry *Mlynek*, każde pole może być: nie zajęte bądź zajęte przez jeden z pionków graczy, a plansza składa się z 24 pól. Takie naiwne szacowanie wynosi więc:

$$\bar{V}_3^{24} = 3^{24} \approx 2,8 \cdot 10^{11} \quad (3.3)$$

a dla *Warcabów angielskich* ($n = 5$, $k = 32$):

$$\bar{V}_5^{32} = 5^{32} \approx 2,32 \cdot 10^{22} \quad (3.4)$$

Jeśli rozważymy gry, w których liczba rodzajów bierek jest większa, omawiane szacowanie zaczyna niebezpiecznie szybko rosnąć, stając się bezużyteczne w praktyce. Założymy, że będą to *Szachy* gdzie dla każdego koloru gracza występuje pięć rodzajów figur oraz pionek. Zbiór wszystkich możliwych stanów pola planszy ma zatem moc 13. Po zastosowaniu dotychczasowego szacowania mamy:

$$\bar{V}_{13}^{64} = 13^{64} \approx 1,96 \cdot 10^{71} \quad (3.5)$$

Składiną wiemy, że liczba dostępnych stanów w *Szachach* nie przekracza 10^{46} . Jak widać błąd takiego szacowania wynosi w tym przypadku, co najmniej 25 rzędów wielkości.

Sytuacja staje się jeszcze bardziej skomplikowana w przypadku gier, w których na jednym polu dozwolone są różne układy, więcej niż jednej z bierek. W grze *Focus* [40] występują tylko dwa rodzaje pionków (po jednym dla każdego z graczy), jednakże na pojedynczym polu mogą znajdować się wieże, składające się maksymalnie z pięciu z nich i to w dowolnych układach.

Jeśli każde pole potraktujemy jako pięć miejsc, z których każde może być w jednym z trzech stanów: {puste, pionek pierwszego gracza, pionek drugiego gracza}, a liczba pól w *Focusie* wynosi 52 otrzymamy astronomiczną liczbę:

$$\bar{V}_3^{52 \cdot 5} = 3^{260} \approx 1,13 \cdot 10^{124} \quad (3.6)$$

Teraz musimy zauważyc, że w grze obowiązują prawa grawitacji. Zatem pionki nie mogą się unosić w przestrzeni, tak więc wewnątrz wieży nie może występować puste miejsce. Możemy zatem zbiór wszystkich możliwych stanów pola planszy $States(5)$ potraktować jako sumę zbiorów:

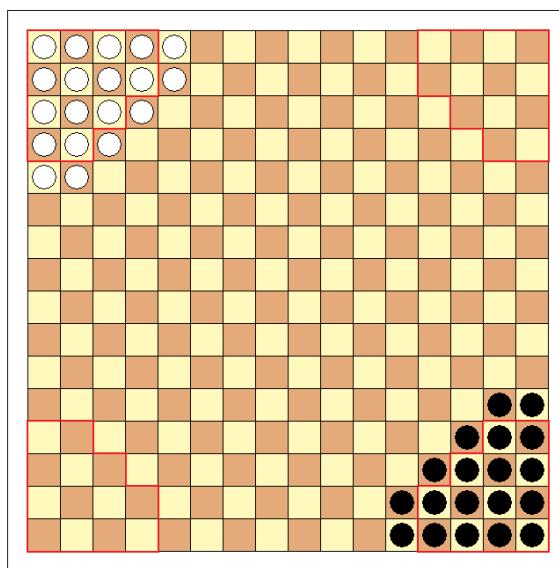
- **z polem pustym** – $States(0)$ o mocy równej 1;
 - **reprezentujących stany pól z n pionkami** pod warunkiem, że: $1 \leq n \leq 5$.
- Do każdej z możliwych wież wysokości $n - 1$ można dodać kolejny n -ty pionek na dwie możliwości, co daje: $|States(n)| = |States(n - 1)| \cdot 2$.

Moc zbioru wszystkich możliwych stanów pola planszy dla gry *Focus* wynosi więc: $States(5) = \sum_{i=0}^5 2^i = 2^6 - 1 = 63$. Zatem nasze szacowanie wyniesie:

$$\bar{V}_{63}^{52} = 63^{52} \approx 3,68 \cdot 10^{93} \quad (3.7)$$

Tym prostym sposobem szacowanie zmalało aż o 31 rzędów wielkości. Dla porównania jest to stosunek mas słońca i tabliczki czekolady.

Istnieją gry, w których liczba pionków nie zmienia się podczas rozgrywki, a na jednym polu może znajdować się maksymalnie jeden pionek. W takich grach górne ograniczenie jest równe liczbie wszystkich permutacji stanów pól gry (pomniejszonej o powtórzenia). Przykładowo dla gry *Halma*¹ (wersja dla dwóch graczy) jest to liczba wszystkich permutacji z powtórzeniami zbioru składającego się z 256 pól, z których na 19 polach znajduje się czarny pion, na 19 biały, natomiast 218 pól jest wolne – rys. 3.1.



RYSUNEK 3.1. Plansza do *Halmy* przedstawiająca pozycję początkową dla dwóch graczy

Taka liczba permutacji dana jest wzorem:

$$P_{n_1, n_2, \dots, n_k} = \frac{(n_1 + n_2 + \dots + n_k)!}{(n_1)! \cdot (n_2)! \cdot \dots \cdot (n_k)!} \quad (3.8)$$

i dla wspomnianych wartości wynosi:

$$P_{19,19,218} = \frac{256!}{19! \cdot 19! \cdot 218!} \approx 1,22 \cdot 10^{56} \quad (3.9)$$

Przestrzeń stanów gry dla większości gier można podzielić na podprzestrzenie spełniające tę cechę (w *Warcabach* – wszystkie stany gry, w których zbita została identyczna liczba pionków gracza białego i gracza czarnego).

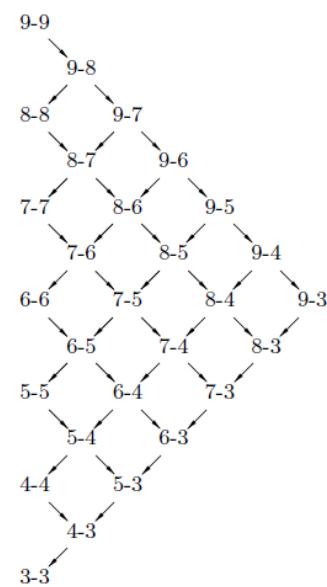
¹Zasady gry *Halma* przedstawiono w dodatku B.10.

Przypadek omówiony dla *Kręgli szachowych* jest szczególny, bo występuje w nim tylko jedna taka podprzestrzeń i jest ona podzbiorem niewłaściwym całej przestrzeni stanów gry. Podział na wspomniane podprzestrzenie zostanie omówiony szczegółowo w następnej sekcji.

3.1.2. Podział przestrzeni stanów gry

Jak zaznaczono w poprzedniej sekcji, jeżeli w grze liczba pionków na planszy ulega zmianie to można przestrzeń stanów traktować jako sumę wszystkich podprzestrzeni o różnej liczbie pionków każda.

Dla gry *Kółko i krzyżyk* będzie to dziesięć podprzestrzeni: $\{0-0, 1-0, 1-1, 2-1, 2-2, 3-2, 3-3, 4-3, 4-4, 5-4\}$ gdzie kolejne ruchy wykonywane są w kolejnych podprzestrzeniach. Teraz górne ograniczenie całej przestrzeni



RYSUNEK 3.2. Podział gry
Młynek [19]

TABLICA 3.1. Podział na podprzestrzenie stanów gry *Kółko i krzyżyk*

<i>Id.</i>	<i>Stan</i>	<i>Liczba pozycji</i>
1.	0-0	1
2.	1-0	9
3.	1-1	72
4.	2-1	252
5.	2-2	480
6.	3-2	1260
7.	3-3	1680
8.	4-3	1260
9.	4-4	630
10.	5-4	126
SUMA:		5770

stanów równe jest sumie ograniczeń wszystkich podprzestrzeni. Jak widać w tab. 3.1 wynosi ono 5770 i jest $\approx 3,411$ raza mniejsze od wyliczonego wcześniej według wzoru (3.2).

Bardziej skomplikowaną grą, dla której można zastosować takie szacowanie, jest *Młynek*. Każdy z graczy dysponuje dziewięcioma pionkami co pozwala na ogranicze-

TABLICA 3.2. Podział na podprzestrzenie stanów gry *Młynek*

	0	1	2	3	4	5	6	7	8	9
0	1	24	276	2.024	10.626	42.504	134.596	346.104	735.471	1.307.504
1	24	552	6.072	42.504	212.520	807.576	2.422.728	5.883.768	11.767.536	19.612.560
2	276	6.072	63.756	425.040	2.018.940	7.268.184	20.593.188	47.070.144	88.256.520	137.287.920
3	2.024	42.504	425.040	2.691.920	12.113.640	41.186.376	109.830.336	235.350.720	411.863.760	594.914.320
4	10.626	212.520	2.018.940	12.113.640	51.482.970	164.745.504	411.863.760	823.727.520	1.338.557.220	1.784.742.960
5	42.504	807.576	7.268.184	41.186.376	164.745.504	494.236.512	1.153.218.528	2.141.691.552	3.212.537.328	3.926.434.512
6	134.596	2.422.728	20.593.188	109.830.336	411.863.760	1.153.218.528	2.498.640.144	4.283.383.104	5.889.651.768	6.544.057.520
7	346.104	5.883.768	47.070.144	235.350.720	823.727.520	2.141.691.552	4.283.383.104	6.731.030.592	8.413.788.240	8.413.788.240
8	735.471	11.767.536	88.256.520	411.863.760	1.338.557.220	3.212.537.328	5.889.651.768	8.413.788.240	9.465.511.770	8.413.788.240
9	1.307.504	19.612.560	137.287.920	594.914.320	1.784.742.960	3.926.434.512	6.544.057.520	8.413.788.240	8.413.788.240	6.544.057.520
								SUMA:	1.43 · 10 ¹¹	

nie przestrzeni stanów do sumy podzbiorów odpowiadających wszystkim możliwym liczbom pionków na planszy przedstawionych w tab. 3.2.

Nagłówek kolumn zawiera liczbę pionków jednego gracza, natomiast nagłówek wierszy – drugiego. Jak widać, oszacowanie zmalało prawie o połowę w porównaniu do wyliczonego według wzoru (3.3).

Jeśli przestrzeń wszystkich stanów jest zbyt duża, aby móc ją w całości przechowywać w pamięci, można ją podzielić na omawiane podprzestrzenie. Pozwoli nam to na rozwiązywanie gry etapami, w szczególności jeśli liczba pozycji końcowych gry jest niewielka. Zalecane jest wtedy rozwiązywanie gry „od końca”, z użyciem mechanizmów analizy wstępnej, szczegółowo omówionej w sekcji 3.2.

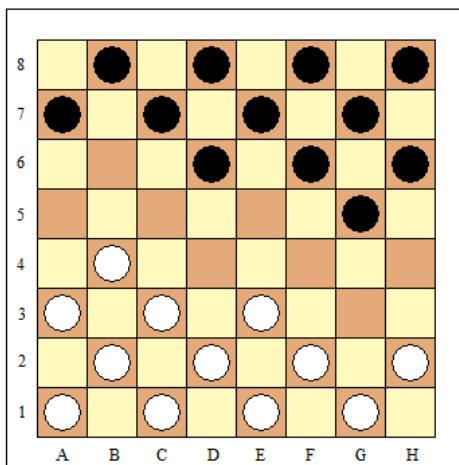
Sytuacja podczas drugiego i trzeciego etapu *Młynka*, w której jeden z przeciwników posiada mniej niż trzy pionki, automatycznie oznacza jego przegrana. Można więc w tej części rozgrywki odrzucić te stany, pozostawiając 28 podprzestrzeni. Dodatkową cenną własnością jest ograniczona do co najwyżej dwóch liczba możliwych przejść z jednej podprzestrzeni do innej. Cechą ta wynika z faktu, że w dwóch ostatnich fazach gry zmiana liczby bierek na planszy następuje tylko w wyniku bicia, a zbite bierki nie wracają do gry. Podział na podprzestrzenie wraz z możliwymi przejściami pomiędzy nimi w grze *Młynek* przedstawiono na rys. 3.2. Łatwo można zauważać, że nie każde ustalenie poprawnej liczby bierek jest możliwe. Wynika to przeważnie z zasad gry. Rozważmy tę sytuację na przykładzie *Warcabów angielskich*. Taką nieosiągalną pozycją może być plansza z pionkami na polu przemiany, które to powinny już

stać się damkami. Po odrzuceniu takich przypadków liczba możliwych stanów w podprzestrzeniach oraz całkowita przestrzeń stanów została przedstawiona w tab. 3.3.

TABLICA 3.3. Złożoność podprzestrzeni stanów gry *Warcaby angielskie* [50]

<i>Liczba pionków</i>	<i>Liczba pozycji</i>
1	120
2	6.172
3	261.224
4	7.092.774
5	148.688.232
6	2.503.611.964
7	34.779.531.480
8	406.309.208.481
9	4.048.627.642.976
10	34.778.882.769.216
SUMA: 1-10	39.271.258.813.439
11	259.669.578.902.016
12	1.695.618.078.654.976
13	9.726.900.031.328.256
14	49.134.911.067.979.776
15	218.511.510.918.189.056
16	852.888.183.557.922.816
17	2.905.162.728.973.680.640
18	8.568.043.414.939.516.928
19	21.661.954.506.100.113.408
20	46.352.957.062.510.379.008
21	82.459.728.874.435.248.128
22	118.435.747.136.817.856.512
23	129.406.908.049.181.900.800
24	90.072.726.844.888.186.880
SUMA: 1-24	500.995.484.682.338.672.639

Jak widać, dzięki takim analizom uzyskano $\approx 46,5$ krotnie mniejsze oszacowanie przestrzeni stanów w stosunku do naiwnego szacowania według wzoru (3.4). Nadal jednak lwnia część stanów gry jest nieosiągalna podczas trwania jakiejkolwiek rozgrywki. Dla przykładu jest to pozycja gdzie jeden z graczy posiada damkę, a żadna bierka przeciwnika nie została jeszcze zbita. Przedostanie się bez bicia przez linię pionków wroga, nawet przy jego pomocy, jest niemożliwe. Takie sytuacje



RYSUNEK 3.3. Przykłady nieosiągalnej pozycji w grze *Warcaby*

można mnożyć, a kolejna przykładowa nieosiągalna pozycja została przedstawiona na rys. 3.3

Znalezienie dokładniejszego górnego ograniczenia lub wzoru, z którego możemy wyliczyć dokładny rozmiar przestrzeni stanów, jest bardzo trudne lub wręcz niemożliwe. Pocieszający jest jednak fakt, że w praktyce wystarczy nam dobre oszacowanie i dokładne wyliczenia analityczne stają się zbędne.

3.1.3. Symetria gry

Sporą redukcję złożoności przestrzeni stanów można uzyskać poprzez wykorzystanie symetrii gry. Wiele gier posiada planszę o więcej niż jednej symetrii. Każda z nich zmniejsza liczbę stanów w przybliżeniu dwukrotnie. Dzieje się tak dlatego, że dwa stany, identyczne względem pewnej symetrii gry, możemy traktować jako jeden. Po obliczeniu wartości tylko jednego z tych stanów, wartości pozostałych są identyczne.

Rozważane symetrie mogą być: osiowe, płaszczyznowe, obrotowe, punktowe. Jeżeli jest ich wiele to bardzo często są one od siebie zależne. Mówiąc inaczej, można pewną symetrię uzyskać ze złożenia innych. Na przykład, symetrie punktowe na płaszczyźnie są złożeniem dwóch symetrii osiowych, a w przestrzeni trzech symetrii płaszczyznowych. Należy więc zdecydować się na pewien ich niezależny zbiór. W skład niego powinny wchodzić takie, że utworzenie stanu symetrycznego oraz sprawdzenie czy dany stan należy do nowej, ograniczonej przestrzeni, jest szybkie i proste w implementacji.

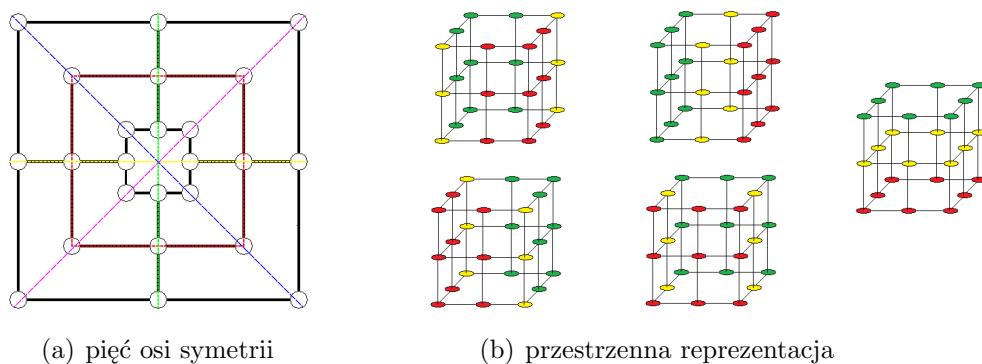
Plansza do gry *Kółko i krzyżyk* posiada cztery osie symetrii (pionową, poziomą i dwie diagonalne), jednak zawsze można jedną z nich złożyć z trzech pozostałych.

Dodatkowo mamy symetrię obrotową względem osi prostopadłej do planszy i przechodzącej przez jej środek. Również ona jest zależna od wspomnianych symetrii osiowych, a w dodatku mniej intuicyjna i przeważnie trudniejsza w implementacji.



RYSUNEK 3.4. Przykłady symetrycznych stanów gry *Kółko i krzyżyk*

W *Kółku i krzyżyku* jest wiele stanów symetrycznych względem jednej bądź większej liczby osi symetrii. Jak wcześniej wspomniano, wpływa to negatywnie na współczynnik redukcji przestrzeni stanów. Na rys. 3.4 zamieszczono ich przykłady.

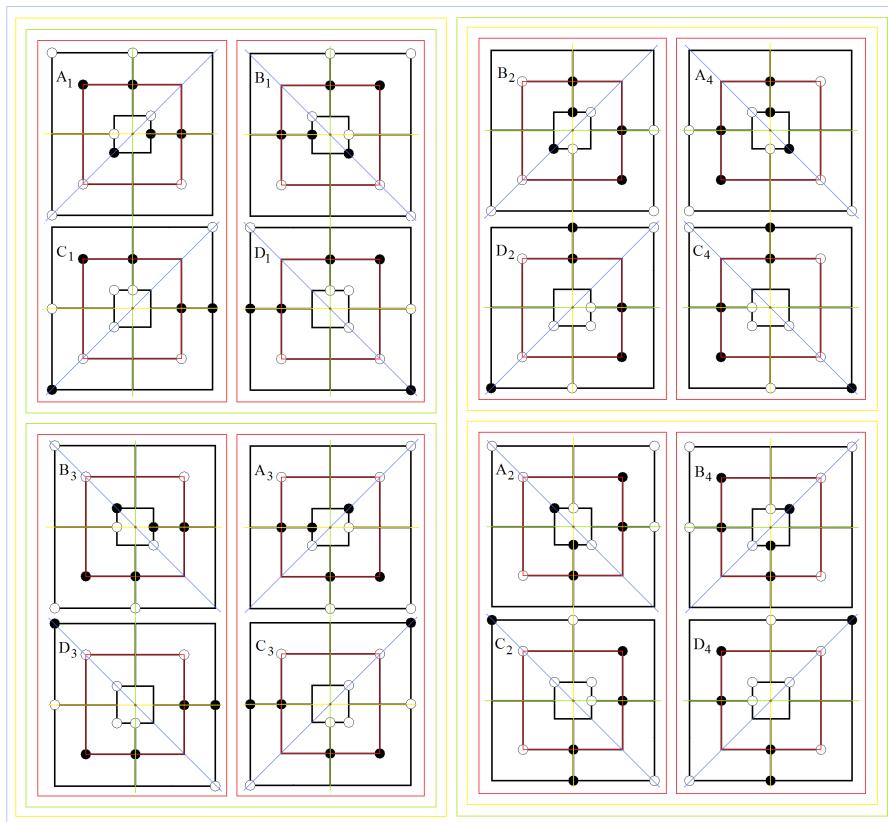


RYSUNEK 3.5. Plansza do gry *Młynek*

Gra *Młynek* posiada aż pięć osi symetrii, widocznych na rys. 3.5(a) oraz jedną obrotową oś symetrii (podobnie jak w przypadku *Kółka i krzyżyka*) prostopadłą do planszy i przechodzącą przez jej środek. Dowolne cztery z tych osi są niezależne. Jednak jedna z nich (czerwona) jest mocno zdeformowana i nie bardzo pasuje do niej to określenie. Jeśli wyobrazić sobie planszę tej gry w przestrzeni to wtedy wszystkie osie stają się poprawnymi płaszczyznami symetrii. Taką przestrzenną reprezentację planszy przedstawiono na rys. 3.5(b). Poprzez wierzchołki w kolorze żółtym przechodzi jedna ze wspomnianych płaszczyzn, rozdzielaając pozostałe wierzchołki na dwie symetryczne podgrupy, oznaczone kolorem zielonym i czerwonym.

Na rys. 3.6 przedstawiam 16 przykładowych symetrycznych stanów tej gry. Kolor ramki określa oś symetrii względem której „odbite” są umieszczone wewnętrznie, układy plansz. Natomiast każdy z czterech ciągów A_1, \dots, A_4 , $B_1, \dots, B_4(\dots)$ to kolejne symetryczne stany gry, obrócone w prawo o 90° względem osi symetrii obro-

towej.



RYSUNEK 3.6. Szesnaście symetrycznych stanów w grze *Mlynek*

Po uwzględnieniu symetrii, przestrzeń stanów gry *Mlynek* zostaje zredukowana do $\approx 7,67 \cdot 10^9$ [19]. W porównaniu do naiwnego szacowania według wzoru (3.3) osiągamy $\approx 36,8$ krotnie mniejsze oszacowanie rozmiaru przestrzeni stanów.

W przypadku niektórych gier liczba ich symetrii może ulegać zmianie podczas rozgrywki. Plansza w szachach, na początku gry, posiada tylko jedną oś symetrii. Jest to spowodowane poruszaniem się pionków gracza tylko w jednym kierunku. Jednak po wyeliminowaniu wszystkich pionków z gry, w następnych posunięciach można uwzględnić wszystkie cztery osie symetrii, z których jak wiemy, każde trzy są niezależne. Taka sytuacja ma miejsce w bazach końcówek szachowych.

3.1.4. Funkcja indeksująca

Bazy danych przechowujące rozwiązane końcówki gier zawierają gigantyczną liczbę stanów. Z jednej strony zależy nam, aby wielkość pamięci zajmowana przez pojedynczy stan była jak najmniejsza. Zaoszczędzenie jednego bitu przekłada się na

oszczędności przestrzeni dyskowej mierzone w gigabajtach. Z drugiej strony, programy rozwiązuające gry charakteryzują się olbrzymią liczbą zapytań do naszej bazy danych, co wymaga szybkiego odszukania właśnie wymaganej pozycji. Musimy zatem każdemu stanowi gry przypisać unikalny indeks oraz zaimplementować funkcję, która będzie potrafiła szybko i jednoznacznie wyznaczyć wartość indeksu na podstawie otrzymanego stanu gry. Funkcja ta nazywana jest funkcją indeksującą. Wartości omawianych indeksów należą do pewnego przedziału liczb całkowitych. Im mniejszy będzie ten przedział, tym mniejsza wielkość pamięci będzie wymagana do przechowania jednej liczby. W idealnym przypadku powinien on rozpoczynać się od zera i kończyć liczbą stanów gry pomniejszoną o jeden. Niestety dla większości gier, jak wcześniej wykazano, już samo obliczenie dokładnej liczby stanów jest bardzo trudne. Tym bardziej stworzenie funkcji indeksującej, która szybko zwracałaby taki indeks jest przeważnie zadaniem bardzo trudnym, jeśli nie niemożliwym do wykonania. A zatem, maksymalna liczba całkowita jaką można zakodować w indeksie, często jest większa od całkowitej liczby dozwolonych stanów gry. Przykładowo, dla *Młynka* udało się stworzyć funkcję indeksującą, mapującą 7.673.759.268 stanów na 9.074.932.579 wskaźników [19]. Łatwo wyliczyć, że współczynnik wykorzystania takiej tablicy wynosi w tym przypadku $\approx 0,846$. Aby na podstawie indeksu można było odtworzyć stan gry, funkcja indeksująca musi być odwracalna, a funkcja odwrotna do niej również powinna być efektywnie zaimplementowana. Opis konstrukcji złożonej funkcji indeksującej dla *Warcabów* i *Szachów* można znaleźć w [32].

3.2. Analiza wsteczna

Analiza wsteczna jest prostą i intuicyjną metodą. Zaczyna ona rozwiązywać grę od jej pozycji końcowych (reprezentujących wygraną, przegrana albo remis), dlatego najlepiej radzi sobie z grami, w których liczba stanów końcowych jest niewielka. Taką własność posiadają gry o naturze zbieżnej, w których przeważnie występuje bicie bierek. Omawiana analiza jest algorytmem o naturze iteracyjnej. Pozwala na silne rozwiązanie gry. Najprostszą jej wersję można ująć w trzech krokach:

1. wygeneruj wszystkie możliwe stany gry¹;
2. przypisz wszystkim pozycjom końcowym etykiety ze zbioru:
 $\{wygrana, remis, przegrana\}$ ²;

¹Do informacji przechowywanych przez stan gry zaliczamy także to, który z graczy ma właśnie wykonać posunięcie.

²Pozycja wygrana dla jednego gracza jest oczywiście przegrana dla jego przeciwnika. Remis nato-

3. przeglądaj kolejne wierzchołki bez etykiety, jeśli:

- aktualny gracz może wykonać przynajmniej jeden ruch do pozycji dla niego wygranej, to oznacz pozycję jako wygraną dla niego;
- wszystkie ruchy prowadzą do pozycji przegranych dla aktualnego gracza, to oznacz pozycję jako przegraną dla niego;
- żaden ruch nie prowadzi do pozycji wygranej dla gracza lub nieznanej, a istnieje ruch do pozycji remisowej to ta pozycja staje się remisowa.

Krok 3 należy wykonywać tak długo, aż wystąpi jeden z możliwych przypadków:

- wartość interesującego nas wierzchołka (np. korzenia) przyjmie znana wartość;
- żadna nowa etykieta wygrana lub przegrana nie została przypisana, wtedy wszystkie wierzchołki z nieznaną wartością oznacz jako remisowe.

Przykładowy algorytm analizy wstecznej w postaci pseudokodu przedstawiono jako alg. 1-4.

Algorytm 1 Analizy wstecznej

```

1: procedure RETROGRADE
2:   GenerateAllPossibleGameStates(root)
3:   changeOccured ← true
4:   while changeOccured do
5:     changeOccured ← RetroStep()
6:   end while
7:   MarkAllUnknownGameStatesAsDraw()
8: end procedure

```

Funkcja generująca wszystkie możliwe stany gry w postaci alg. 2 jest używana jedynie, gdy mamy możliwość wygenerowania wszystkich ruchów, jakie mogą pojawić się w grze rozpoczynającej się od pozycji *root*. W bazach końcówek przeważnie postępuje się inaczej, generując wszystkie ustawienia o odpowiedniej liczbie pionków.

Podczas korzystania z algorytmu analizy wstecznej można natrafić na trzy grupy problemów:

miast dotyczyć obu graczy jednocześnie.

Algorytm 2 Generujący wszystkie możliwe stany gry

```

1: procedure GENERATEALLPOSSIBLEGAMESTATES(root)
2:   stack.Push(root)
3:   while stack.Count > 0 do
4:     node ← stack.Pop()
5:     if DataBase.Contains(node) then
6:       continue
7:     end if
8:     if EndPosition(node) then
9:       MarkPosition(node)
10:    else
11:      for all child in node.children do
12:        stack.Push(child)
13:      end for
14:    end if
15:    DataBase.Add(node)
16:   end while
17: end procedure

```

- **problem ogromnej przestrzeni stanów gry.** Najlepiej jeżeli w grze występuje jakiś naturalny czynnik, pozwalający podzielić ją na mniejsze podprzestrzenie. Najczęściej jest to liczba bierek na planszy. W sekcji 3.1.2 przedstawiono taki podział dla gry *Młynek* (tab. 3.2 i rys. 3.2) oraz dla gry *Warcabły angielskie* (tab. 3.3). Dla każdej liczby n -bierek budowana jest osobna baza danych wykorzystująca wyniki z bazy ($n - 1$) i wcześniejszych. Wartość każdego ruchu powodującego zmniejszenie liczby pionków o jeden lub więcej jest wyliczona we wcześniej rozwiązanych bazach dla mniejszej liczby pionków. Pozwala to na użycie systemów rozproszonych, każdy rozwiązujący osobną podprzestrzeń.
- **problem czasu wykonania.** Związany jest z wielokrotnym (do kilkudziesięciu iteracji) przeglądaniem wcześniej poetykietowanych wierzchołków. Usprawnieniem jest przechowywanie rozwiązanych wierzchołków w kolejce, a w każdej iteracji analiza wierzchołków rodzicielskich¹ z tej kolejki. Rozwiązani rodzice są dodawani do kolejki a wierzchołki, których wszyscy rodzice zostali rozwiązani, usuwane.

¹Pozycje bezpośrednio prowadzące do tych wierzchołków.

Algorytm 3 Kroku analizy wstępnej

```
1: function RETROSTEP
2:   changeOccured ← false
3:   for all node in DataBase do
4:     if node.value = unknown then
5:       drawChildExist ← false
6:       unknownChildExist ← false
7:       for all child in node.children do
8:         if child.value = lose then
9:           node.value ← win
10:          changeOccured ← true
11:          break
12:        else if child.value = unknown then
13:          unknownChildExist ← true
14:        else if child.value = draw then
15:          drawChildExist ← true
16:        end if
17:      end for
18:      if node.value = unknown then
19:        if unknownChildExist then
20:          continue
21:        else if drawChildExist then
22:          node.value ← draw
23:        else
24:          node.value ← lose
25:          changeOccured ← true
26:        end if
27:      end if
28:    end if
29:  end for
30:  return changeOccured
31: end function
```

Algorytm 4 Oznaczający nieznane stany jako remisowe

```

1: procedure MARKALLUNKNOWNGAMESTATESAsDRAW
2:   for all node in DataBase do
3:     if node.value = unknown then
4:       node.value ← draw
5:     end if
6:   end for
7: end procedure

```

- **problem olbrzymiej liczby instrukcji wejścia-wyjścia.** Tutaj szczególnie wrażliwe są systemy rozproszone. Dlatego podczas pierwszej iteracji na dysku lokalnym buforowane są wszystkie pozycje, które wymagały odwołania do innej jednostki systemu rozproszonego.

Dokładne informacje jak poradzono sobie z budową bazy danych końcówek *Warcabów angielskich* można znaleźć w [32].

3.3. Algorytmy *depth-first search*

3.3.1. Algorytm Minimax

Już na początku lat pięćdziesiątych Claude E. Shannon zauważył, że komputer dysponujący nieograniczoną pamięcią i mocą obliczeniową mógłby perfekcyjnie grać w *Szachy* [57]. Wystarczy dla pozycji startowej rozważyć wszystkie możliwe posunięcia, z kolei dla każdego z tych posunięć rozważyć wszystkie możliwe dla niego posunięcia przeciwnika itd. Ponieważ każda rozgrywka w szachach składa się ze skończonej liczby posunięć¹ kiedyś w końcu muszą wystąpić pozycje końcowe, których wartość będzie ustalona. Założymy, że wartości tych pozycji P przypisywane poprzez funkcję $Evaluate(node)$ będą następujące:

$$Evaluate(P) = +1 \text{ -- dla pozycji wygrywającej}$$

$$Evaluate(P) = 0 \text{ -- dla pozycji remisowej}$$

$$Evaluate(P) = -1 \text{ -- dla pozycji przegrywającej}$$

Cofając się po powstałym drzewie przypisujemy każdemu z możliwych stanów gry odpowiadającą mu wartość. Ustalamy ją na podstawie wartości wcześniej wyliczonych dla każdego z możliwych posunięć w danym stanie gry. To wyliczenie inaczej

¹Trzykrotne powtórzenie się tej samej pozycji oznacza remis.

odbywa się dla stanów gry gracza rozpoczynającego, a inaczej dla jego przeciwnika. Wartość każdego ze stanów, w którym ruch należy do rozpoczynającego będzie równa maksymalnej wartości któregokolwiek z możliwych do wykonania ruchów. Jest to oczywiste ponieważ gracz wybierze w tym momencie najlepsze dla siebie posunięcie. Dla każdego ze stanów, w którym ruch należy do przeciwnika należy przypisać wartość minimalną z możliwych ruchów. A to dlatego, że w *Szachach* każdy z graczy dysponuje pełną informacją i przeciwnik wykonuje posunięcia najgorsze dla gracza rozpoczynającego. W ten sposób możemy cofnąć się, aż do korzenia drzewa gry, czyli do pozycji początkowej i określić jej wartość. Możemy więc teoretycznie określić czy pozycja startowa *Szachów* jest dla rozpoczynającego wygrywająca, przegrywająca czy remisowa. Oczywiście zakładając, że obydwa będą wykonywali optymalne posunięcia.

W ten sposób sposób działa algorytm *Minimax*. Jest algorytmem typu *depth-first search*, dlatego że w każdym kolejnym kroku bierze do rozwinienia wierzchołek z poziomu głębszego o jeden. Dopiero po natrafieniu i przypisaniu wartości liściowi odpowiadającemu pozycji końcowej w grze zaczyna się cofać, ustalając wartości wierzchołków na mniejszych głębokościach. Algorytm ten ma naturę rekurencyjną i w takiej postaci został przedstawiony jako alg. 5.

Algorytm 5 Minimax

```

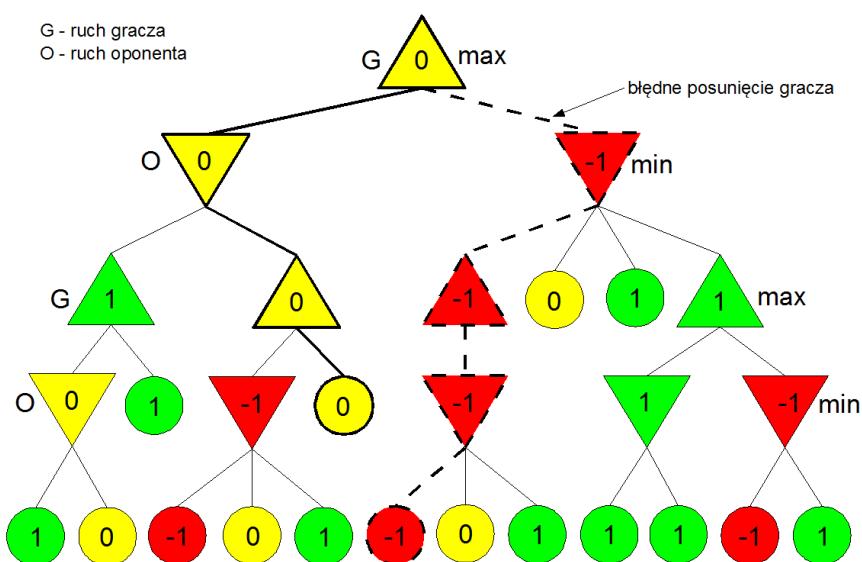
1: procedure MINIMAX(node)
2:   if IsTerminal(node) then
3:     return Evaluate(node)
4:   end if
5:   if opponentTurn then
6:     return  $\text{Min}_{N \in \text{Children}(node)} \text{Minimax}(N)$ 
7:   else
8:     return  $\text{Max}_{N \in \text{Children}(node)} \text{Minimax}(N)$ 
9:   end if
10: end procedure

```

Zauważmy, że w przedstawionej postaci algorytm nie będzie działał poprawnie jeżeli reguły gry pozwalają na wielokrotne powtarzanie się pewnej pozycji gry. Taka sytuacja ma miejsce w *Szachach*. Rozwiązaniem jest przechowywanie w pamięci listy wierzchołków wchodzących w skład ścieżki od korzenia do aktualnego wierzchołka. Funkcja *IsTerminal(node)* powinna sprawdzać czy wierzchołek nie występuje na tej

liście więcej razy niż pozwalają na to zasady gry. Jeżeli tak się stanie to zostaje on uznany za wierzchołek terminalny, a funkcja *Evaluate(node)* powinna mu przypisać wartość remisową, czyli 0.

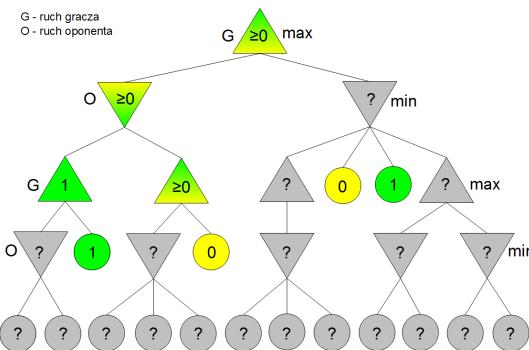
Na rys. 3.7 przedstawiono drzewo *minimaksowe* obrazujące działanie algorytmu *Minimax*. Jego pozycja startowa została wyliczona jako remisowa. Pogrubioną linią zaznaczono przebieg optymalnej rozgrywki. Łatwo zauważać, że jeśli gracz rozpoczynający partię wykona alternatywne (błędne) posunięcie (linia przerywana) to przy optymalnej grze przeciwnika będzie musiał przegrać partię.



RYSUNEK 3.7. Drzewo minimaksowe

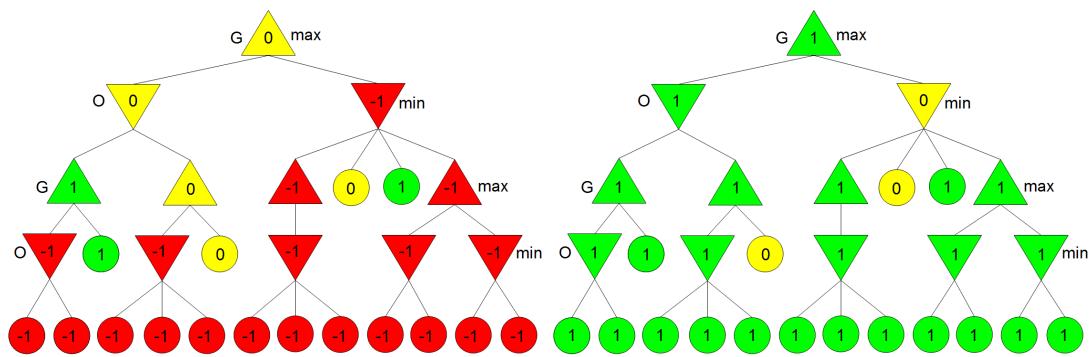
Jeżeli drzewo rozwiązywanego problemu jest zbyt głębokie, można spróbować „przyciąć” je do pewnej głębokości i zobaczyć czy uda nam się pozyskać jakąś informację na temat rozwiązywanego problemu. Przy takim podejściu funkcja *Is-Terminal(node)* powinna sprawdzać, czy podany wierzchołek to końcowa pozycja gry lub czy dalsze jego rozwijanie nie spowoduje przekroczenia limitu głębokości poszukiwań. Limit ten nie musi być zresztą zadany w postaci ustalonego ograniczenia na odległość od korzenia, jednakowego dla wszystkich węzłów. Możemy bowiem „bardziej obiecujące rejony drzewa” rozwijać głębiej posługując się heurystycznymi funkcjami oceny reprezentującymi wiedzę ekspercką. Oczywiście decydując się na „przycięcie” drzewa gry możemy ograniczyć rozważaną przestrzeń stanów do zbioru nie wystarczającego do jednoznacznego określenia wartości korzenia. Często jednak możliwe jest uzyskanie informacji częściowej. Rys. 3.8 przedstawia takie drzewo i pomimo że nie znamy wartości żadnego z wierzchołków na głębokości odcięcia

wiemy, że rozwiązanie jest z przedziału $[0,1]$.



RYSUNEK 3.8. Przycięte drzewo minimaksowe

Taki przedział możemy znaleźć wyszukując dolne i górne ograniczenia osobno. Pierwsze z nich znajdziemy poprzez wpisanie minimalnej wartości (-1 – przegrana) do każdego wierzchołka, którego wartość nie jest znana, a nie może on być dalej rozwijany wskutek ograniczenia głębokości drzewa. Teraz wartości naszego przyjętego drzewa przedstawione na rys. 3.9(a). Wyliczona wartość korzenia to (0 – remis) i równa jest wartości dolnego ograniczenia. Drugie, górne ograniczenie znajdujemy analogicznie wpisując zamiast wartości minimalnych, maksymalne, czyli (1 – wygrana) rys. 3.9(b).



(a) z wpisanymi wartościami minimalnymi (b) z wpisanymi wartościami maksymalnymi

RYSUNEK 3.9. Przycięte drzewo minimaksowe

W najlepszym przypadku może się zdarzyć, że wartości dolnego i górnego ograniczenia będą sobie równe, a wtedy mamy jednoznacznie określone rozwiązanie.

Złożoność obliczeniowa algorytmu *Minimax* wynosi $O(b^d)$, gdzie b to uśredniony współczynnik rozgałęzienia, a d to głębokość drzewa. Złożoność pamięciowa wynosi $O(b \cdot d)$, jeżeli w jednym kroku działania algorytmu rozwijane są wszystkie wierzchołki potomne, albo $O(d)$ jeżeli rozwijany jest tylko przeszukiwany wierzchołek.

3.3.2. Algorytm NegaMax

W opisany wcześniej algorytmie w kolejnych krokach wybieraliśmy albo wartość maksymalną jeśli znajdowaliśmy się w węźle gracza, który rozpoczął rozgrywkę, albo wartość minimalną w węźle przeciwnika. W algorytmie NegaMax połączono te dwie procedury w jedną, wykorzystując fakt opisany równaniem 3.10

$$\forall a_1, a_2, \dots, a_n \in \Re : \min(a_1, a_2, \dots, a_n) = -\max(-a_1, -a_2, \dots, -a_n) \quad (3.10)$$

Podstawiając za $\min[\max(a_1, a_2, \dots, a_n), \dots, \max(x_1, x_2, \dots, x_n)]$ równanie 3.10 otrzymujemy kolejne równanie 3.11

$$\begin{aligned} & \min[\max(a_1, a_2, \dots, a_n), \dots, \max(x_1, x_2, \dots, x_n)] = \\ & -\max[-\max(a_1, a_2, \dots, a_n), \dots, -\max(x_1, x_2, \dots, x_n)] \end{aligned} \quad (3.11)$$

Zatem wystarczy w każdym wywołaniu funkcji *Negamax* wybierać tylko wartość maksymalną, a wartości zwracane przez rekurencyjne wywołania tej funkcji negować. Stąd też wzięła się nazwa tego algorytmu.

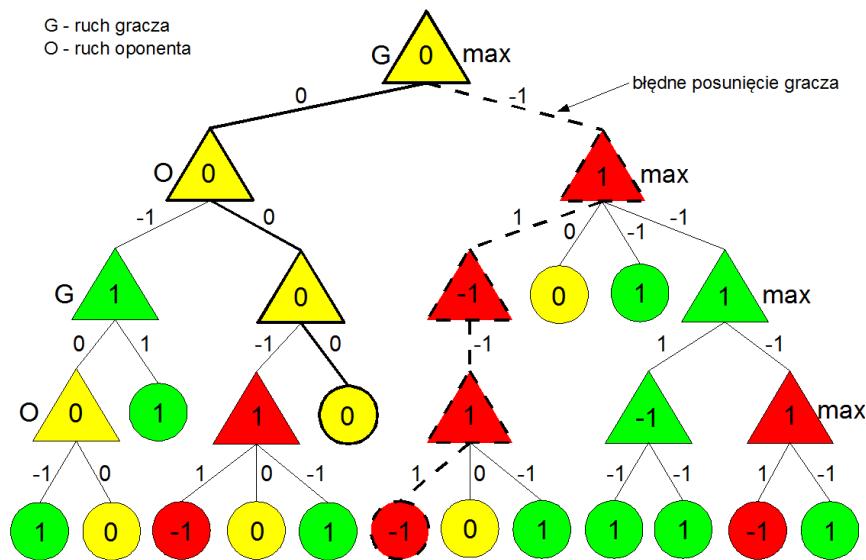
Na rys. 3.10 przedstawiono drzewo *negamaxowe* dla analogicznej sytuacji jak na rys. 3.7. Widać, że wszystkie wierzchołki są wierzchołkami maksymalizującymi, natomiast wartości we wcześniejszych, wewnętrznych wierzchołkach minimalizujących są przeciwnie do ich odpowiedników w drzewie *Minimaksozym*¹.

Algorytm *Negamax* został przedstawiony jako alg. 6. Wynik jego działania jest identyczny z wynikiem algorytmu *Minimax* a jest on krótszy w implementacji.

3.3.3. Przycinanie $\alpha-\beta$

Dotychczas zaprezentowane algorytmy w swojej pierwotnej formie mogą zostać zastosowane jedynie do rozwiązania bardzo prostych gier, takich jak *Kółko i krzyżyk* oraz jako teoretyczne wprowadzenie do ich bardziej rozwiniętych wersji. Największym problemem wspomnianych algorytmów jest konieczność przeglądnienia całego

¹Dlatego też funkcja *Evaluate(node)* musi ulec lekkiej modyfikacji i w wierzchołkach oznaczających posunięcia przeciwnika musi zwracać wartości przeciwnie.



RYSUNEK 3.10. Drzewo negamaksowe

Algorytm 6 Negamax

```

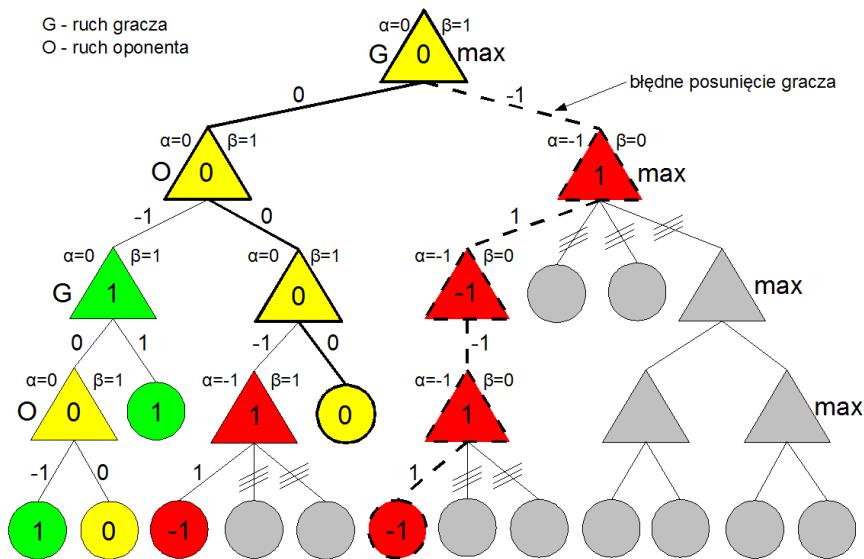
1: procedure NEGAMAX(node)
2:   if IsTreminal(node) then
3:     return Evaluate(node)
4:   end if
5:   return  $\text{Max}_{N \in \text{Children}(node)} - \text{Negamax}(N)$ 
6: end procedure

```

drzewa gry, które jak wiemy z rozdziału 2, jest olbrzymie. Pierwsze usprawnienie, pozwalające na praktyczne wykorzystanie algorytmu dla bardziej skomplikowanych problemów opublikował Donald Knuth i Ronald Moore w 1975. Nosi ono nazwę *przycinanie α - β* [52] i oparte jest o średniowieczną zasadę *Brzytwy Ockham'a*. Jego istota polega na umiejętności określeniu poddrzew drzewa gry, które nie mają wpływu na aktualnie wyliczony jej wynik i rezygnacji z ich przeglądania. To przycinanie wymaga wprowadzenia dwóch dodatkowych zmiennych α i β :

- α – zawiera ocenę najlepszej dotychczas możliwej do realizacji strategii gracza rozpoczęjącego (ograniczenie dolne możliwego wyniku).
- β – zawiera ocenę najgorszej dotychczas możliwej strategii dla gracza rozpoczęjącego do jakiej może doprowadzić przeciwnik (ograniczenie górne możliwego wyniku).

Inaczej mówiąc, jeśli algorytm *Minimax* wywołany w celu oceny pewnego drzewa zwróci wartość x , algorytm z przycinaniem użyty z parametrami $\alpha \leq \beta$ dla tego samego drzewa zwróci wynik będący medianą z $\{\alpha, x, \beta\}$ przypuszczalnie przeglądając mniejszą liczbę wierzchołków (redukcja jest tym silniejsza, im węższy jest przedział $[\alpha, \beta]$).



RYSUNEK 3.11. Drzewo negamaksowe z odcinaniem $\alpha-\beta$

Na rys. 3.11 przedstawiono drzewo *Negamax'owe* (sytuacja analogiczna jak na rys. 3.7 i 3.10) z odciętymi za pomocą $\alpha-\beta$ gałęziami. Dodatkowo zamieszczono wartości zmiennych α i β jakie ustalały się po wykonaniu algorytmu dla każdego z wierzchołków drzewa.

Wierzchołki oznaczane na szaro nigdy nie zostaną odwiedzone przez algorytm. Dzieje się tak, ponieważ wartości zwrócone przez wcześniej odwiedzonego potomka były większe od współczynnika β co oznaczało, że przeszukiwanie pozostałych wierzchołków nie miało sensu. A to dlatego, że przeciwnik i tak obierze wcześniej już znalezioną, lepszą dla niego drogę.

Algorytm *Minimax* wzbogacony o funkcję przycinania $\alpha-\beta$ został przedstawiony jako alg. 7 a jego wersja *negamaxowa* jako alg. 8. Współczynniki α i β przyjmują inicjalnie odpowiednio najmniejszą i największą możliwą wartość oceny gry. W przypadku wspomagania się funkcją oceniającą jest to przeważnie przedział liczb rzeczywistych. Wtedy $\alpha = -\infty$ a $\beta = \infty$, dla deterministycznej oceny gry wystarczy przyjąć: $\alpha = -1$ a $\beta = 1$.

Algorytm 7 Minimax z przycinaniem $\alpha-\beta$

```

1: procedure MINIMAXALFABETA(node,  $\alpha$ ,  $\beta$ )
2:   if IsTerminal(node) then
3:     return Evaluate(node)
4:   end if
5:   if opponentTurn then
6:     for all child in node.children do
7:        $\beta \leftarrow \text{Min}(\beta, \text{MinimaxAlfaBeta}(\text{child}, \alpha, \beta))$ 
8:       if  $\alpha \geq \beta$  then
9:         return  $\alpha$ 
10:      end if
11:    end for
12:    return  $\beta$ 
13:   else
14:     for all child in node.children do
15:        $\alpha \leftarrow \text{Max}(\alpha, \text{MinimaxAlfaBeta}(\text{child}, \alpha, \beta))$ 
16:       if  $\alpha \geq \beta$  then
17:         return  $\beta$ 
18:       end if
19:     end for
20:     return  $\alpha$ 
21:   end if
22: end procedure

```

Współczynniki α i β możemy traktować jako znany *a priori* przedział, w którym znajduje się nasze rozwiązanie. Możemy bowiem dysponować inną wiedzą, która pozwoli nam już wstępnie ograniczyć ten przedział. Jeżeli, okaże się że został on źle dobrany, to w wyniku dostaniemy wartość α lub β i będziemy ją mogli wykorzystać podczas kolejnych obliczeń jako górne (wynik = α) albo dolne (wynik = β) ograniczenie. Przykładowo, jeżeli pewna analiza zwróci oszacowanie wartości jakiegoś wierzchołka w postaci przedziału $[0, 1]$ (tzn. wiemy, że jej oszacowanie dolne wynosi 0), musimy spróbować rozwiązać tę pozycję znajdując oszacowanie górne podobnie jak na rys. 3.9(b) używając zamiast procedury *Minimax* przycinania z wartościami $\alpha = 0, \beta = 1$.

Algorytm 8 Negamax z przycinaniem $\alpha-\beta$

```
1: procedure NEGAMAXALFABETA(node,  $\alpha$ ,  $\beta$ )
2:   if IsTerminal(node) then
3:     return Evaluate(node)
4:   end if
5:   for all child in node.children do
6:     val  $\leftarrow$  -NegamaxAlfaBeta(child,  $-\beta$ ,  $-\alpha$ )
7:     if val  $\geq \beta$  then
8:       return  $\beta$ 
9:     end if
10:    if val  $> \alpha$  then
11:       $\alpha \leftarrow$  val
12:    end if
13:   end for
14:   return  $\alpha$ 
15: end procedure
```

3.4. Struktury reprezentujące wiedzę o grze

Dotychczas opisane algorytmy typu *depth-first search* przechowywały informacje o stanie gry w kolejnych rekurencyjnych wywołaniach programu czyli na stosie systemowym, który zazwyczaj jest niewielki. Takich spoczywających na stosie odwołań było nie więcej niż wynosi głębokość przeszukiwanego drzewa gry, czyli nie więcej od liczby posunięć w rozwiązywanej grze. Dodatkowo w celu detekcji powtórzeń w pamięci przechowywana była lista wierzchołków wchodzących w skład ścieżki prowadzącej od korzenia drzewa do aktualnie wyliczanego wierzchołka. Lista ta, również nie mogła być dłuższa od głębokości drzewa gry.

Algorytmy typu *best-first search* opisane w sekcji 3.5 do przechowywania wiedzy o grze wymagają bardziej złożonych struktur danych. Są to przeważnie struktury generyczne zawierające w każdym swoim węźle jeden stan gry. Wybór odpowiedniej z nich narzucony jest poprzez naturę danej gry. W następnej kolejności zostaną omówione trzy najczęściej stosowane:

- drzewo AND/OR
- DAG - digraf acykliczny AND/OR
- DCG - digraf cykliczny AND/OR

3.4.1. Drzewo skierowane AND/OR

Drzewo AND/OR to skierowane drzewo ukorzenione, w skład którego wchodzą dwa typy wierzchołków:

- **OR** – jest sumą logiczną swoich wierzchołków potomnych¹;
- **AND** – jest iloczynem logicznym swoich wierzchołków potomnych.

Wierzchołki typu OR zawsze oznaczają posunięcie gracza rozpoczynającego (korzeń drzewa zawsze jest typu OR). Wierzchołki typu AND oznaczają posunięcia przeciwnika. Każdemu z tych wierzchołków przypisana jest wartość ze zbioru $\{ \text{true}, \text{false}, \text{unknown} \}$. Wartość ta określa czy stan gry przechowywany przez wierzchołek drzewa jest dla gracza rozpoczynającego: wygrywający, przegrywający albo czy algorytm przeszukujący nie zdążył jeszcze określić tej wartości. Wartości te są przypisywane w następujący sposób:

wartość *true*:

- jeśli wynika to bezpośrednio z reguł gry i wygrał gracz do którego należą posunięcia z węzłów OR, jest to wtedy wierzchołek terminalny;
- dla wierzchołka typu OR, jeśli przynajmniej jeden jego potomek jest wygrywający, wartość pozostałych potomków nie ma wtedy znaczenia i można je pominąć;
- dla wierzchołka typu AND, jeśli wszystkie wierzchołki potomne są wygrywające.

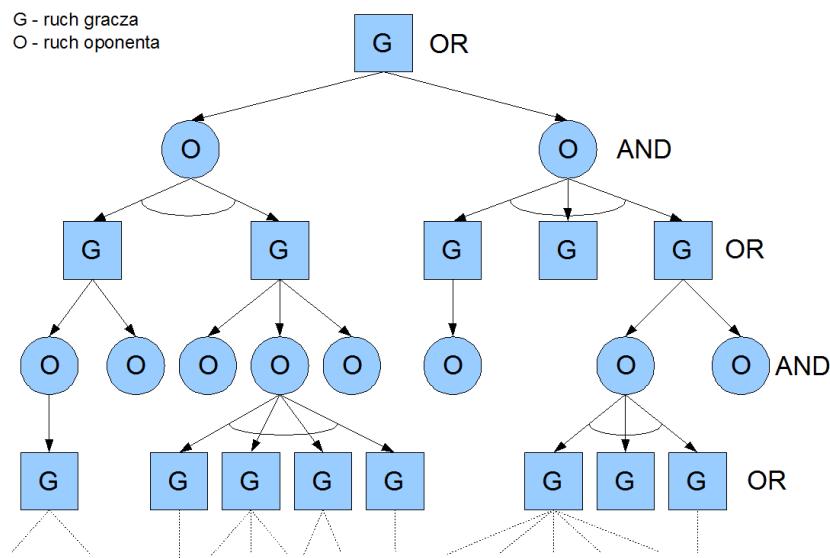
wartość *false*:

- jeśli wynika to bezpośrednio z reguł gry i wygrał gracz do którego należą posunięcia z węzłów AND, jest to wtedy wierzchołek terminalny;
- dla wierzchołka typu OR, jeśli wszystkie wierzchołki potomne są przegrywające;
- dla wierzchołka typu AND, jeśli przynajmniej jeden jego potomek jest przegrywający, wartość pozostałych potomków nie ma wtedy znaczenia i można je pominąć.

wartość *unknown*:

¹Przy założeniu, że wygraną gracza rozpoczynającego oznaczymy jako logiczne 1, a przegrana jako logiczne 0.

- inicjalnie przypisujemy korzeniowi drzewa;
- podczas dołączania nowych wierzchołków potomnych, przeważnie dodawanych przez algorytm przeszukujący na podstawie wyników generatora ruchów.
- dla wierzchołka typu OR, jeśli przynajmniej jeden potomek ma wartość nieznaną¹ i żaden z potomków nie jest wygrywający;
- dla wierzchołka typu AND, jeśli przynajmniej jeden potomek ma wartość nieznaną² i żaden z potomków nie jest przegrywający.



RYSUNEK 3.12. Drzewo AND/OR

Przykład takiego drzewa pokazuje na rys. 3.12. Może być ono stosowane do rozwiązywania gier, w których każda pozycja jest osiągalna tylko przez jedną unikalną sekwencję posunięć. Jednak, bodaj żadna z popularnych gier nie posiada tej własności. Nie oznacza to, że tego typu drzewa są bezużyteczne. Są one szeroko stosowane w grach nie spełniających powyższego warunku. W takim przypadku należy się jednak liczyć z możliwością redundancji³ skutkującej nadmiernym zużyciem pamięci, a w przypadku istnienia cykli, z możliwością zapętlenia się algorytmu.

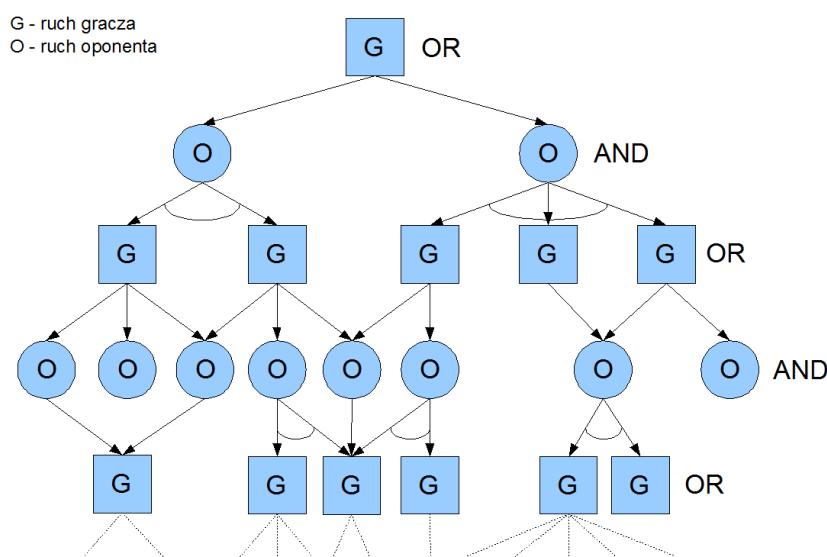
¹Dotyczy to również przypadku kiedy nie wszystkie wierzchołki potomne zostały rozwinięte.

²Analogicznie jak w przypadku wierzchołka OR może to być również nie rozwinięty wierzchołek potomny.

³Istnieniem identycznych poddrzew posiadających jako korzeń wierzchołek z identycznym stanem gry.

3.4.2. DAG – digraf acykliczny AND/OR

W skład digrafa acyklicznego AND/OR, w skrócie DAG⁴, wchodzą analogicznie jak w przypadku drzew AND/OR, dwa typy wierzchołków AND i OR. Każdemu z nich przypisana jest wartość ze zbioru $\{true, false, unknown\}$ i wartość ta przypisywana jest identycznie jak dla drzewa AND/OR, jednak DAG różni się od drzewa AND/OR tym, że może zawierać wierzchołki stopnia wejściowego większego niż 1.



RYSUNEK 3.13. Digraf acykliczny AND/OR

Przykład takiego digrafa został pokazany na rys. 3.13. Może być stosowany do rozwiązywania gier, w których podczas trwania rozgrywki nie może wystąpić powtórzenie dowolnego stanu gry. Warunek ten jest spełniony np. dla gier, w których z każdym posunięciem zwiększa się lub zmniejsza liczba pionów. Przykładem takich gier są: *Pentomino*, *Quarto* oraz wszystkie gry z rodziny gier połączeniowych, szerzej opisane w sekcji 2.3.

3.4.3. DCG – digraf cykliczny AND/OR

W skład digrafa cyklicznego AND/OR w skrócie DCG¹ wchodzą analogicznie jak w przypadku drzew AND/OR i DAG, dwa typy wierzchołków AND i OR. Tutaj również każdemu z wierzchołków przypisana jest wartość ze zbioru $\{true, false, unknown\}$. Przypisywana jest identycznie jak dla drzewa AND/OR z uwzględnieniem

⁴Akronim od *Directed Acyclic Graph*.

¹Akronim od *Directed Cyclic Graph*.

dodatkowej sytuacji. Owa sytuacja ma miejsce wtedy kiedy za pomocą wspomnianych zasad nie da się pewnym wierzchołkom przypisać żadnej wartości ze zbioru: $\{\text{true}, \text{false}\}$. Mówimy, że występuje wówczas sytuacja remisowa². Jednak wierzchołki AND, OR nie są przystosowane do reprezentowania takich sytuacji. Należy zatem przypisać im jedną z wartości $\{\text{true}, \text{false}\}$. Zależnie od tej reprezentacji, dla sytuacji remisowej różnie interpretujemy wartość korzenia:

- **reprezentacja poprzez wartość *false*** – jeżeli wierzchołkom remisowym przypisujemy wartości *false* to wartość korzenia równa *true* oznacza wygraną gracza rozpoczynającego, natomiast wartość *false* jest górnym ograniczeniem i oznacza albo jego przegrana albo remis;
- **reprezentacja poprzez wartość *true*** – jeżeli wierzchołkom remisowym przypisujemy wartości *true* to wartość korzenia równa *true* jest dolnym ograniczeniem i oznacza albo wygraną gracza rozpoczynającego albo remis, natomiast wartość *false* oznacza jednoznacznie jego przegrana.

Jak łatwo zauważyc, dwukrotne poetykietowanie DAG za każdym razem z inną reprezentacją pozycji remisowych, pozwala na jednoznaczne określenie wartości korzenia. Jeżeli mamy szczęście to już za pierwszym razem możemy poznać jednoznaczny wynik.

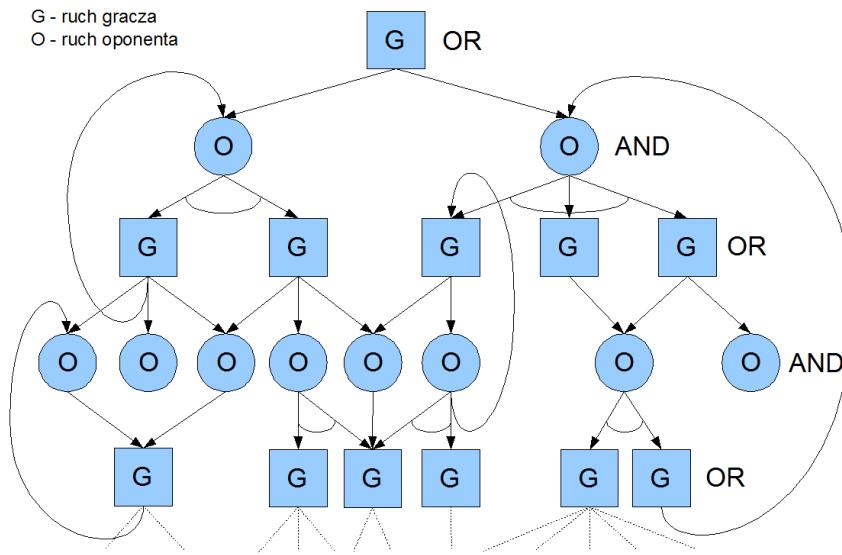
W swojej budowie DCG różni się od DAG jedynie możliwością zawierania cykli. Przykładowy DCG przedstawiono na rys. 3.14. Pasuje on idealnie do modelowania gier w których podczas trwania rozgrywki może wystąpić powtórzenie dowolnego stanu gry. Warunek ten występuje w większości gier, na przykład w: *Młynku*, *Szachach*, *Warcabach*, *Go*.

3.5. Algorytmy *best-first search*

Algorytmy typu *best-first search* starają się wybierać do rozwinięcia tylko najbardziej optymalne wierzchołki¹, tj. selektywnie rozwijać różne obszary drzewa zgodnie z heurystycznymi regułami mającymi na celu rozwiązanie gry po zbadaniu jak najmniejszej liczby wierzchołków. Skuteczność takiego podejścia została potwierdzona między innymi podczas rozwiązywania gier *Czwórki* [2], *Qubic*, *Go-Moku* [3]. Rozwinięcie wierzchołka składa się z dwóch faz:

²Zgodnie z twierdzeniem Zermelo (sekcja 2.2), gracz może w nieskończonej liczbie posunięć opóźniać swoją przegrana.

¹Główna różnica pomiędzy algorytmami *best-first search* polega właściwie na różnych kryteriach oceny optymalności wierzchołka.



RYSUNEK 3.14. Digraf cykliczny AND/OR

1. **rozszerzenie** (z ang. *expand*) – polega na wygenerowaniu wszystkich wierzchołków potomnych rozwijanego wierzchołka;
2. **wyliczenie** (z ang. *evaluate*) – dotyczy wszystkich wierzchołków potomnych rozwijanego wierzchołka wygenerowanych w fazie 1. Polega na przypisaniu każdemu z nich wartości ze zbioru: $\{ \text{true}, \text{false}, \text{unknown} \}$ według zasad szczegółowo opisanych w 3.4.1.

Faza wyliczenia może przebiegać na dwa różne sposoby:

- **natychmiastowe wyliczenie** – każdy wierzchołek po wygenerowaniu jest natychmiast wyliczany. Podczas inicjalizacji drzewa zostaje utworzony i wyliczony korzeń i dopóki będzie posiadał wartość *unknown*, dopóty w każdym kroku wybierany jest najbardziej dowodzący wierzchołek² (pojęcie najbardziej dowodzącego wierzchołka zostało szczegółowo opisane w następnej sekcji). Następnie, wszystkie jego wierzchołki potomne są, zaraz po wygenerowaniu wyliczane;
- **opóźnione wyliczenie** – wartość wierzchołka jest po raz pierwszy wyliczana w momencie wybrania go najlepszym wierzchołkiem. Podczas inicjalizacji drzewa

²Wierzchołek jest już wcześniej wyliczony i musi to być wartość *unknown*.

zostaje utworzony niewyliczony korzeń i dopóki będzie posiadał wartość *unknown*, dopóty w każdym kroku wybierany jest najlepszy wierzchołek, a następnie wyliczany. Jeśli otrzyma wartość *unknown* to zostają wygenerowane wszystkie jego wierzchołki potomne, ale bez wyliczania ich wartości.

Wersja algorytmu z opóźnionym wyliczaniem powinna być stosowana w przypadkach, gdzie procedura wyliczająca jest czasochłonna. W obydwu wersjach tej procedury liczbom dowodzącej i obalającej rozwijanego wierzchołka zostają przypisane wartości 1. W przypadku gdy łatwo możemy uzyskać informację o liczbie wierzchołków potomnych rozwijanego wierzchołka możemy zastosować pewne udoskonaleń. Polega ono na tym, że dla wierzchołka typu OR liczbę dowodzącą inicjujemy wartością 1 (wystarczy udowodnić tylko jeden z wierzchołków potomnych), a liczbę obalającą inicjujemy liczbą wierzchołków potomnych tego wierzchołka (należy udowodnić wszystkie jego wierzchołki potomne). W przypadku wierzchołka typu AND wartości te przypisujemy odwrotnie. Takie udoskonalenie pozwala na zmniejszenie liczby wygenerowanych wierzchołków o połowę, jest to wynik eksperymentów autora.

Podczas tworzenia drzewa AND/OR mogą w nim występować cztery rodzaje wierzchołków:

- **wewnętrzne** – to takie, które posiadają przynajmniej jednego potomka.
- **terminalne** – to takie, które przedstawiają pozycję końcową gry. Są to wierzchołki rozwiązane co oznacza, że posiadają jedną z wartości ze zbioru: {*true*, *false*}. Nie mogą posiadać potomków.
- **graniczne** (wyliczone)¹ – to takie, których wartość została wyliczona jako *unknown*², a wierzchołki potomne nie zostały jeszcze rozwinięte.
- **graniczne** (niewyliczone)³ – to takie, których wartość nie została wyliczona, a wierzchołki potomne nie zostały jeszcze rozwinięte.

3.5.1. Algorytm Proof-Number Search

Algorytm Proof-Number Search, w skrócie nazywany *pn-search*, wymaga dwóch rodzajów informacji z dziedziny rozwiązywanego problemu:

¹Występują przy zastosowaniu natychmiastowego wyliczania.

²W przeciwnym przypadku, gdyby ich wartość została wyliczona do *true* lub *false* stałyby się wierzchołkami terminalnymi.

³Występują przy zastosowaniu opóźnionego wyliczania.

- Pierwsza, to liczba wierzchołków potomnych każdego z analizowanych wierzchołków. Wartość ta jest zwykle zwracana przez generator ruchów danej gry.
- Druga, to wartość ze zbioru $\{true, false, unknown\}$ jaką posiada każdy analizowany wierzchołek. Wartość ta jest zwykle zwracana przez funkcję oceniającą koniec rozgrywki.

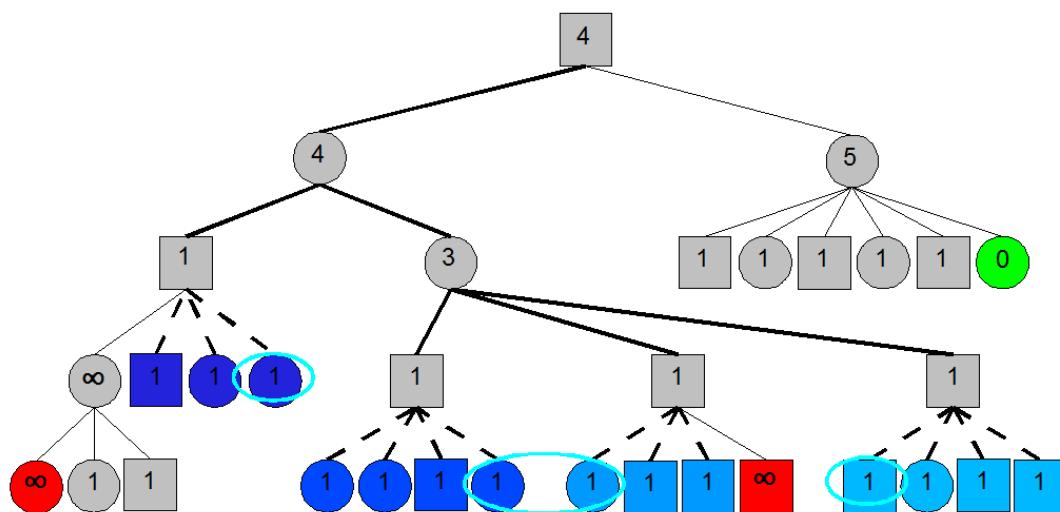
Nie oznacza to jednak, że dodatkowa wiedza, znana *a priori*, nie może zostać użyta. Stosuje się ją w usprawnieniach tego algorytmu jako dodatkowe źródło informacji podczas wyboru najlepszego wierzchołka.

Celem działania algorytmu *pn-search* jest rozwiązanie korzenia, czyli ustalenie jednej z dwóch wartości: *true* lub *false*, inaczej mówiąc udowodnienie lub obalenie drzewa. Jeżeli wartości *true* oznaczają wygraną gracza rozpoczynającego to udowodnienie drzewa jest jednoznaczne z posiadaniem przez niego strategii wygrywającej. W przypadku gier bez remisów, gdzie wartości *false* oznaczają jego przegrany, obalenie drzewa oznacza, że to przeciwnik posiada strategię wygrywającą. W grach mogących kończyć się remisami użycie algorytmu jest bardziej złożone. Jeżeli wartością *false* oznaczymy przegrany lub remis wówczas obalenie drzewa oznacza istnienie strategii „unikania przegranej” dla przeciwnika, a więc jesteśmy w pozycji przegranej lub remisowej. Natomiast po oznaczeniu remisów i wygranych przez *true* dowiedzenie drzewa oznacza, że gracz rozpoczynający jest w pozycji wygranej lub remisowej. Dokładne rozwiązanie gry może więc wymagać dwukrotnego, skutecznego wykonania procedury. W celu ustalenia jednej z wymienionych wartości, algorytm poszukuje zbioru dowodzącego bądź zbioru obalającego. Zbiór dowodzący (obalający) – jest to zbiór wierzchołków granicznych, których udowodnienie (obalenie), udowadnia (obala) całe drzewo.

- **liczba dowodząca** – jest to moc najmniejszego zbioru dowodzącego;
- **liczba obalająca** – jest to moc najmniejszego zbioru obalającego;

Rys. 3.15 przedstawia drzewo AND/OR z zaznaczonym przykładowym, minimalnym zbiorem dowodzącym. Każdy wierzchołek zawiera liczbę dowodzącą. Dla każdego wierzchołka granicznego liczba ta równa jest 1, ponieważ wymaga on udowodnienia wyłącznie siebie samego. Wierzchołki terminalne dowodzące (zielone) posiadają liczbę dowodzącą równą 0, ponieważ nie trzeba już niczego dla nich udowadniać. W przeciwnieństwie do nich wierzchołki terminalne obalające (czarne) posiadają liczbę dowodzącą równą ∞ , ponieważ nie istnieje już możliwość ich udowodnienia.

Dla wierzchołków wewnętrznych liczba dowodząca zależy od ich typu.

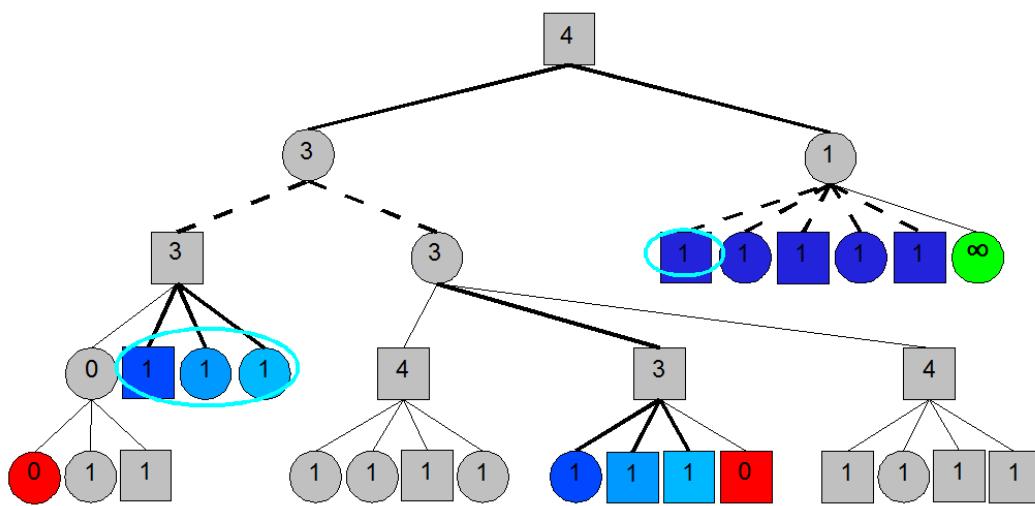


RYSUNEK 3.15. Drzewo AND/OR z liczbami dowodzącymi

- **Wierzchołki wewnętrzne typu OR** – przedstawiają posunięcie gracza rozpoczynającego, dla którego poszukiwany jest zbiór dowodzący. Gracz może wykonać posunięcie najkorzystniejsze dla niego. Zatem wierzchołek OR zawiera minimum z liczb dowodzących swoich potomków.
- **Wierzchołki wewnętrzne typu AND** – przedstawiają posunięcie przeciwnika. Może on więc wykonać dowolne posunięcie. Zatem wierzchołek ten będzie udowodniony dopiero, jeśli każde z posunięć przeciwnika będzie dowodzące. Reasumując, wierzchołek AND zawiera sumę liczb dowodzących wszystkich swoich potomków.

W ten sposób przypisując wszystkim wierzchołkom granicznym wartość 1, a następnie wyliczając wartości liczb dowodzących ich przodków, możemy określić liczbę dowodzącą dla korzenia. W tym przypadku wynosi ona 4. Oznacza to, że minimalna liczba wierzchołków granicznych, których udowodnienie udowodni drzewo, równa jest 4. Aby odnaleźć te wierzchołki należy poczawszy od korzenia, poruszać się w głąb drzewa. W każdym kroku wybieramy zbiór wierzchołków potomnych, których suma liczb dowodzących równa jest liczbie dowodzącej rozpatrywanego wierzchołka. Na rys. 3.15 zilustrowano to pogrubionymi liniami, linie przerywane oznaczają alternatywne w dokonywaniu wyboru na danym poziomie. Taka możliwość wyboru oznacza istnienie wielu minimalnych zbiorów dowodzących. W opisany przypadku jest ich $3 \cdot 4 \cdot 3 \cdot 4 = 144$.

Wszystkie one są jednak równoliczne, a ich moc wynosi 4. Kolorem niebieskim zaznaczono wszystkie wierzchołki należące do któregokolwiek z minimalnych zbiorów



RYSUNEK 3.16. Drzewo AND/OR z liczbami obalającymi

dowodzących. Rys. 3.16 przedstawia analogiczną sytuację przykładowego, minimalnego zbioru obalającego dla tego samego drzewa AND/OR. Liczby w wierzchołkach granicznych również posiadają wartość 1. Wartości wierzchołków terminalnych są teraz wyznaczone według odwrotnej niż poprzednio reguły, ponieważ to wierzchołek obalony nie wymaga dodatkowych wyliczeń, a wierzchołek dowiedziony nie może zostać obalony. Odwrotnie natomiast następuje wyliczanie wartości liczb obalających dla wierzchołków wewnętrznych. W wierzchołkach OR są one sumą liczb obalających wszystkich potomków. W wierzchołkach AND jest to minimum z wszystkich liczb obalających potomków. Tutaj zbiorów minimalnych również jest więcej niż jeden bo $2 \cdot 5 = 10$, a przykładowy z nich zaznaczono elipsami.

Ponieważ nie zawsze wiemy jaka będzie wartość korzenia, nie wiemy również jakiego zbioru powinniśmy szukać. Najlepiej jeśli moglibyśmy rozwijać obydwa zbiory równocześnie. Szczęśliwie okazuje się, że zawsze istnieje wierzchołek, który będzie należał zarówno do minimalnego zbioru dowodzącego, jak i do obalającego, zwany „najbardziej dowodzącym wierzchołkiem”.

Twierdzenie 3.1. Jeżeli korzeń drzewa AND/OR nie został rozwiązany to zawsze istnieje najbardziej dowodzący wierzchołek.

Dowód.

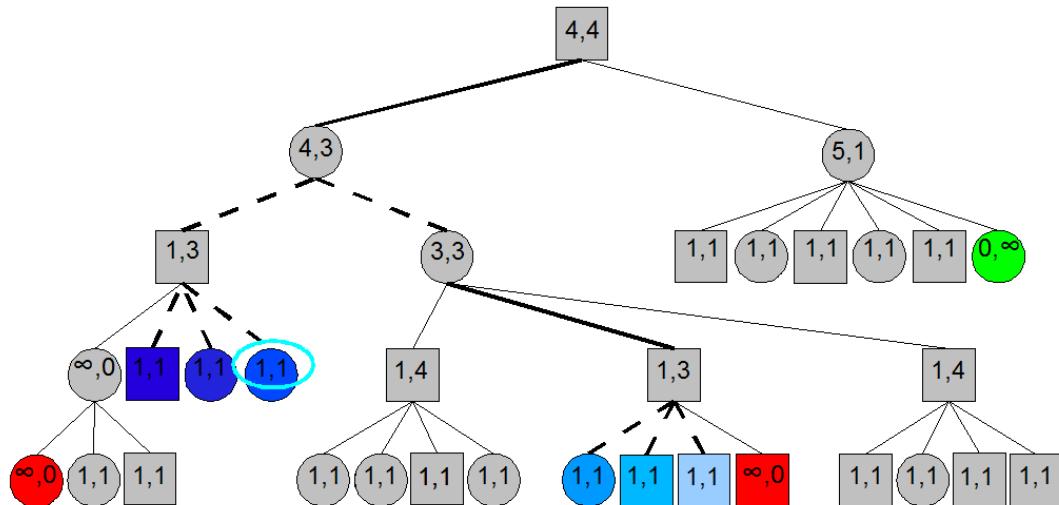
- **krok podstawowy** – Dla każdego granicznego wierzchołka v , zbiór jednoelementowy zawierający wierzchołek v jest zarazem jedynym zbiorem dowodzącym i jedynym zbiorem obalającym wierzchołek v . Przekrój tych dwóch zbiorów również zawiera wierzchołek v , z czego wynika, że nie jest zbiorem pustym.

- **krok indukcyjny** – Przypuśćmy, że nasze założenie zostało udowodnione dla wszystkich wierzchołków potomnych v_1, \dots, v_n , wewnętrznego wierzchołka v typu AND. Aby obalić wierzchołek v , wystarczy aby jeden z jego potomków był obalony. Niech $ob(v_x)$ będzie zbiorem obalającym pewnego wierzchołka v_x posiadającym najmniejszą moc pośród wszystkich zbiorów obalających wierzchołków potomnych v . Wierzchołek v_x jest również minimalnym zbiorem obalającym v . Aby udowodnić wierzchołek v , wszystkie jego wierzchołki potomne muszą być udowodnione. Niech $dow(v_i)$ dla $(1 \leq i \leq n)$ będzie własnym minimalnym zbiorem dowodzącym dla każdego z wierzchołków potomnych v_i . $\cup_{i=1}^n dow(v_i)$ jest minimalnym zbiorem dowodzącym v . Zatem $ob(v_x)$ jest minimalnym zbiorem obalającym i zawiera się w minimalnym zbiornie dowodzącym v . Ponieważ $ob(v_x)$ i $dow(v_x)$ są minimalnymi zbiorami dowodzącymi i obalającymi wierzchołka v_x , ich przekrój zgodnie z założeniem indukcyjnym, nie jest zbiorem pustym. Zatem $ob(v)$ i $dow(v)$ posiadają niepusty przekrój.

Dla wierzchołka typu OR dowód przeprowadza się analogicznie.

□

Możemy więc w jednym kroku algorytmu rozwinać obydwa zbiory, wybierając właśnie ten wierzchołek. Będziemy go nazywać **najbardziej dowodzącym wierzchołkiem**.



RYSUNEK 3.17. Drzewo AND/OR z najbardziej dowodzącym wierzchołkiem

Rys. 3.17 przedstawia sposób, w jaki można znaleźć najbardziej dowodzący wierzchołek dla wcześniej opisywanego drzewa. Każdy z wierzchołków zawiera teraz zarówno

liczbę dowodzącą (po prawej stronie) jak i obalającą (po lewej stronie). Zaczynając od korzenia zależnie od rodzaju wierzchołka, w którym się znajdujemy należy:

- dla wierzchołka **OR** – wybrać dowolny potomny wierzchołek zawierający identyczną liczbę dowodzącą;
- dla wierzchołka **AND** – wybrać dowolny potomny wierzchołek zawierający identyczną liczbę obalającą.

W rozważanym przykładzie, aż sześć wierzchołków może być kandydatami na najbardziej dowodzący wierzchołek. Zgodnie z wcześniejszymi spostrzeżeniami należą one do przekroju pewnych minimalnych zbiorów: dowodzącego i obalającego.

Algorytm 9 Proof-Number Search

```

1: procedure PNSEARCH(root, maxNodes)
2:   Evaluate(root)
3:   SetProofAndDisproofNumbers(root)
4:   while root.proof ≠ 0 AND root.disproof ≠ 0 AND CountNodes() ≤ maxNodes
   do
5:     mostProvingNode ← SelectMostProving(root)
6:     DevelopNode(mostProvingNode)
7:     UpdateAncestors(mostProvingNode)
8:   end while
9:   if root.proof = 0 then
10:    root.value ← true
11:   else if root.disproof = 0 then
12:    root.value ← false
13:   else
14:    root.value ← unknown
15:   end if
16: end procedure

```

Algorytm *pn-search* przedstawiony został jako alg. 9. Pierwszą procedurą wykonywaną dla korzenia, później również często wykorzystywana, jest *Evaluate(node)*. Przypisuje ona danemu wierzchołkowi wartość *true* – jeśli jest to wierzchołek dowiedziony, *false* – jeśli wierzchołek jest obalony. W przeciwnym przypadku wierzchołek dostaje wartość *unknown*. Wartości te są określone na podstawie stanu gry przechowywanego w danym węźle. Przykładowo, jeśli stan zawiera końcową pozycję szachową, wygraną

dla białych, bądź pozycję z bazy danych końcówek szachowych, która posiada strategię wygrywającą dla białych, procedura $Evaluate(node)$ przypisuje temu wierzchołkowi wartość *true*.

Algorytm 10 Ustalający liczby proof i disproof

```

1: procedure SETPROOFANDDISPROOFNUMBERS(node)
2:   if node.expanded then
3:     if node.type = and then
4:       node.proof  $\leftarrow \sum_{N \in Children(\textit{node})} N.\text{proof}$ 
5:       node.disproo  $\leftarrow \min_{N \in Children(\textit{node})} N.\text{disproo}$ 
6:     else if node.type = or then
7:       node.proof  $\leftarrow \min_{N \in Children(\textit{node})} N.\text{proof}$ 
8:       node.disproo  $\leftarrow \sum_{N \in Children(\textit{node})} N.\text{disproo}$ 
9:     end if
10:    else if node.evaluated then
11:      if node.value = false then
12:        node.proof  $\leftarrow \infty$ 
13:        node.disproo  $\leftarrow 0$ 
14:      else if node.value = true then
15:        node.proof  $\leftarrow 0$ 
16:        node.disproo  $\leftarrow \infty$ 
17:      else if node.value = unknown then
18:        node.proof  $\leftarrow 1$ 
19:        node.disproo  $\leftarrow 1$ 
20:      end if
21:    else
22:      node.proof  $\leftarrow 1$ 
23:      node.disproo  $\leftarrow 1$ 
24:    end if
25: end procedure
  
```

Kolejnym krokiem jest wywołanie procedury $SetProofAndDisprooNumbers(node)$ na korzeniu. Została ona przedstawiona jako alg. 10. Jej działanie jest zależne od rodzaju wierzchołka.

- **dla wierzchołków rozwiniętych** – oblicza liczbę dowodzącą albo obalającą wierzchołka, w zależności jakiego jest typu: AND czy OR.

- **dla wierzchołków wyliczonych** – w przypadku wierzchołków udowodnionych, przypisuje liczbie dowodzącej wartość 0, a liczbie obalającej wartość ∞ . Dla wierzchołków obalonych liczba dowodząca otrzymuje wartość ∞ , a obalająca 0. W przypadku wierzchołków wyliczonych do wartości *unknown* obydwie liczby otrzymują wartość 1.
- **dla wierzchołków niewyliczonych** – przypisuje wartość 1 zarówno liczbie dowodzącej jak i obalającej.

Główna część algorytmu *pn-search* odbywa się w pętli pomiędzy liniami: 4 – 8. Pętla ta wykonuje się dopóki spełnione są trzy warunki:

1. korzeń nie został udowodniony;
2. korzeń nie został obalony;
3. liczba wierzchołków przechowywanych w pamięci nie przekroczyła zadeklarowanej, pozwala to na określenie górnej granicy pamięci przeznaczonej do wykorzystania przez algorytm.

Wewnątrz głównej pętli wykonywane są kolejno trzy instrukcje: wyszukiwanie najbardziej dowodzącego wierzchołka, jego rozwinięcie oraz uaktualnianie wartości jego przodków. Poniżej kolejno opisano ich działanie.

- **Instrukcja wybierająca najbardziej dowodzący wierzchołek**, przedstawiona jako alg. 11, to funkcja zwracająca taki właśnie wierzchołek. Jest pojedynczą pętlą, w której zaczynając od korzenia i kierując się liczbami dowodzącymi w wierzchołkach OR oraz liczbami obalającymi w wierzchołkach AND, wybierany jest kolejny potomek. Pętla kończy się jeśli trafimy na wierzchołek graniczny, który zwracany jest tą przez funkcję.
- **Procedura rozwijająca wierzchołek**, może wystąpić w dwóch wersjach (obydwie wersje przedstawiono jako alg. 12 i 13):
 - **W wersji z opóźnionym wyliczaniem** na początku wierzchołek v jest wyliczany, poprzez wykonanie procedury *Evaluate(node)*. Jeżeli nie okaże się on wierzchołkiem terminalnym, w kolejnym kroku algorytm wykonuje instrukcję *GenerateAllChildren(node)*, która jest powiązana z generatorem ruchów gry i tworzy listę kolejnych możliwych (zgodnych z zasadami danej gry) stanów tej gry. Każdy taki stan generuje jeden

Algorytm 11 Wybierający najbardziej dowodzący wierzchołek

```

1: function SELECTMOSTPROVING(node)
2:   while node.expanded do
3:     if node.type = or then
4:       i  $\leftarrow$  0
5:       while node.children[i].proof  $\neq$  node.proof do
6:         i  $\leftarrow$  i+1
7:       end while
8:     else if node.type = and then
9:       i  $\leftarrow$  0
10:      while node.children[i].disproof  $\neq$  node.disproof do
11:        i  $\leftarrow$  i+1
12:      end while
13:    end if
14:    node = node.children[i]
15:  end while
16:  return node
17: end function

```

Algorytm 12 Rozwijający wierzchołek (*wersja z natychmiastowym wyliczaniem*)

```

1: procedure DEVELOPNODE(node)
2:   GenerateAllChildren(node)
3:   for all child in node.children do
4:     Evaluate(child)
5:     SetProofAndDisproofNumbers(child)
6:   end for
7: end procedure

```

wierzchołek potomny dla v . Następnie, poprzez wykonanie procedury *SetProofAndDisproofNumbers(node)* na każdym węźle potomnym obliczane są jego liczby: dowodząca i obalająca.

- **W wersji z natychmiastowym wyliczaniem** procedura *Evaluate(node)* nie musi być wykonywana na wierzchołku v , ponieważ już wcześniej został on wyliczony. Natomiast jest ona wykonywana na każdym, nowo dodanym węźle potomnym wierzchołka v .

- Ostatnim krokiem jest wywołanie **procedury uaktualniającej wartości**

Algorytm 13 Rozwijający wierzchołek (*wersja z opóźnionym wyliczaniem*)

```

1: procedure DEVELOPNODE(node)
2:   Evaluate(node)
3:   if node.value = unknown then
4:     GenerateAllChildren(node)
5:     for all child in node.children do
6:       SetProofAndDisproofNumbers(child)
7:     end for
8:   end if
9: end procedure

```

Algorytm 14 Uaktualniający przodków

```

1: procedure UPDATEANCESTORS(node)
2:   while node ≠ NIL do
3:     SetProofAndDisproofNumbers(node)
4:     node ← node.parent
5:   end while
6: end procedure

```

liczb dowodzących i obalających przodków danego wierzchołka, przedstawionej jako alg. 14. Jej działanie polega na wykonywaniu wcześniej opisanej instrukcji *SetProofAndDisproofNumbers(node)* na wszystkich wierzchołkach potomnych poczynając od rodzica, a kończąc na korzeniu.

Po zakończeniu głównej pętli algorytm przypisuje korzeniowi wartość *true* jeśli został udowodniony¹, wartość *false* jeśli został obalony², lub wartość *unknown* co oznacza, że zostały przekroczone zadeklarowane zasoby pamięciowe.

3.5.2. Usuwanie rozwiązań poddrzew

Jednym z usprawnień algorytmu *pn-search*, pozwalającym na zmniejszenie wykorzystywanej przez niego pamięci, jest usuwanie rozwiązań poddrzew. To usprawnienie polega na usuwaniu z pamięci rozwiązań wierzchołków zaraz po aktualizacji wartości ich rodziców poprzez procedurę *UpdateAncestors(node)*.

Aby wykazać poprawność omawianego usprawnienia zauważmy, że w drzewie algorytmu *pn-search* dowolny wierzchołek wpływa na proces przeszukiwania tylko

¹Jeśli wartość jego liczby dowodzącej jest równa zero.

²Jeśli wartość jego liczby obalającej jest równa zero.

w dwóch przypadkach:

- jeśli znajdzie się na drodze od korzenia do najbardziej dowodzącego wierzchołka;
- jeśli podczas wykonywania instrukcji *SetProofAndDisproofNumbers(node)* jego liczby dowodząca i obalająca mają wpływ na analogiczne liczby u jego rodzica.

Teraz należy pokazać, że:

- w drzewie z nieroziązaniem korzeniem nigdy nie znajdzie się rozwiązany wierzchołek na drodze do najbardziej dowodzącego wierzchołka.
- liczby dowodząca i obalająca rozwiązań wierzchołków albo natychmiast przyczyniają się do rozwiązania rodzica, albo nie mają wpływu na ich wartość.

Udowadniając pierwsze twierdzenie należy zauważyć, że funkcja wyszukująca najbardziej dowodzący wierzchołek startuje od korzenia, a jak założyliśmy, korzeń nie jest jeszcze rozwiązany. Jak wiemy każdy nieroziązany wierzchołek posiada skończone wartości liczb dowodzącej i obalającej, różne od zera. Ponieważ w każdym kroku wybierany jest taki wierzchołek, którego wartość liczby dowodzącej (dla wierzchołka AND) lub obalającej (dla wierzchołka OR) jest równa wartości odpowiadającej jej liczby u potomka, wartości tych liczb dla kolejno wybieranych wierzchołków będą skończone i różne od zera. Reasumując, wszystkie wierzchołki na drodze do najbardziej dowodzącego wierzchołka będą nieroziązane.

W drugim twierdzeniu należy zauważyć, że rozwiązań wierzchołek o wartości *true* posiada wartość liczby dowodzącej równą 0, a obalającej ∞ . Wpływ takiego wierzchołka zależy od typu rodzica:

- **jeśli rodzic jest wierzchołkiem typu OR** to wartość jego liczby dowodzącej jest równa najmniejszej z wszystkich wartości liczb dowodzących swoich potomków czyli w tym przypadku 0. Rodzic zostaje więc natychmiast udowodniony.
- **jeśli rodzic jest wierzchołkiem typu AND** to wartość jego liczby dowodzącej jest równa sumie wszystkich liczb dowodzących swoich potomków czyli w tym przypadku wartość 0 nie ma wpływu na rodzica. Natomiast wartość liczby obalającej równa ∞ zawsze będzie wyższa od pozostałych, skończonych wartości liczb dowodzących reszty rodzeństwa czyli również pozostanie bez wpływu na rodzica.

Dla wierzchołka rozwiązanego o wartości *false* sytuacja jest dokładnie odwrotna. Liczby dowodząca i obalająca wynoszą odpowiednio $(\infty, 0)$ i wpływ takiego wierzchołka na wierzchołek typu OR jest identyczny, jak w przypadku wierzchołka o wartości *true* na wierzchołek AND i vice versa.

Jak widać wszystkie rozwiązane wierzchołki albo przyczyniły się do rozwiązania ich rodziców, albo nie mają na ich wartość żadnego wpływu. Również w przyszłości nigdy nie będą miały takiego wpływu i nie pojawią się na ścieżce prowadzącej do najbardziej dowodzącego wierzchołka. Można więc bezkarnie usunąć je z pamięci.

3.5.3. Algorytm pn^2 -search

Innym sposobem zaoszczędzenia pamięci jest rozszerzenie algorytmu *pn-search* zwane *pn²-search* [3, 11]. Polega on na podzieleniu tego algorytmu na dwa poziomy. Na pierwszym poziomie, który będziemy oznaczać jako (pn_1) dla każdego wierzchołka podczas jego rozwijania, wywoływany jest, w miejsce instrukcji *GenerateAllChildren(node)*, algorytm *pn-search* drugiego poziomu, który będziemy oznaczać jako (pn_2) . Algorytm (pn_2) wywoływany jest z ograniczeniem na rozmiar drzewa. Ograniczenie to nie jest stałe i jest równe wielkości dotychczas rozwiniętego drzewa na poziomie (pn_1) zwiększonej o jeden¹. Podczas takiego rozwijania wierzchołek przekazywany z poziomu (pn_1) jest na poziomie (pn_2) traktowany jako korzeń i może zostać standardowo wyliczony do jednej z wartości: $\{true, false, unknown\}$. I co najważniejsze liczby dowodząca i obalająca wszystkich wierzchołków potomnych rozwianego wierzchołka, wyliczone na poziomie (pn_2) , pozostają bez zmian na poziomie (pn_1) , a ich poddrzewa usunięte z pamięci.

Algorytm 15 Rozwijający wierzchołek w algorytmie pn^2 -search

```

1: procedure DEVELOPNODE(node)
2:   PNSearch(node, ComputeMaxNodes())
3:   for all child in node.children do
4:     child.DeleteSubtree()
5:   end for
6: end procedure

```

Usprawnienie to wymaga w stosunku do oryginalnego *pn-search* jedynie zmienionej funkcji *DevelopNode(node)*, która została przedstawiona jako alg. 15. Używana w nim procedura *ComputeMaxNodes()* zwraca górne ograniczenie liczby wierzchołków drzewa

¹To zwiększenie pozwala na rozwinięcie wierzchołków potomnych dla drzewa zawierającego jedynie korzeń, ma to miejsce podczas pierwszego wywołania algorytmu (pn_2) .

na poziomie (pn_2). To ograniczenie może być równe, jak wcześniej zaznaczono dotyczącej liczbie wierzchołków powiększonej o jeden. Jednak lepsze rezultaty otrzymamy mnożąc liczbę wierzchołków drzewa na poziomie (pn_1) przez współczynnik podany wzorem 3.12 [11].

$$f(x) = \frac{1}{1 + e^{\frac{(a-x)}{b}}} \quad (3.12)$$

Jest on funkcją rosnącą przyjmującą wartości z przedziału $[0, 1]$, gdzie x oznacza liczbę wierzchołków drzewa na poziomie (pn_1). Kształt oraz szybkość narastania tej funkcji modelujemy odpowiednio dobierając parametry a i b . Zastosowanie tego współczynnika wynika z faktu, że na początku kiedy drzewo jest niewielkie nie ma sensu głębokie przeszukiwanie na poziomie (pn_2). A to dlatego, że dla małych drzew prawdopodobieństwo rozwiązania któregokolwiek z wierzchołków granicznych jest niewielkie. Dodatkowo niepotrzebnie opóźniany jest rozwój drzewa na pierwszym poziomie.

Teoretycznie usprawnienie $pn^2 - search$ pozwala dla drzewa rozmiaru N na pierwszym poziomie odwiedzić $\sim \frac{1}{2} \cdot N^2$ wierzchołków. Jak udowodniono empirycznie [3] algorytm ten pozwala na kwadratowy wzrost liczby przeglądanych wierzchołków przy identycznym zużyciu pamięci.

3.5.4. Redukcja czasu wykonania

Podczas każdego wyszukiwania najbardziej dowodzącego wierzchołka przeglądane są wszystkie wierzchołki, począwszy od korzenia, a skończywszy na szukanym wierzchołku. Podczas uaktualniania przodków ta sama droga przebywana jest z powrotem. Ponieważ procedury te są wykonywane w głównej pętli programu, czas ich wykonania ma istotny wpływ na czas wykonania całego algorytmu. Posłużymy się dwoma obserwacjami:

- jeśli podczas procedury uaktualniania przodków żadna z wartości liczb dowodzącej i obalającej nie ulegnie zmianie, możemy przerwać dalsze uaktualnianie, ponieważ nie wprowadzi ono już żadnych zmian;
- jeśli pewien wierzchołek leży na drodze prowadzącej do najbardziej dowodzącego wierzchołka, a jego wartości liczb dowodzącej i obalającej nie uległy zmianie podczas przeprowadzania procedury uaktualniania przodków, to wierzchołek ten nadal leży na tej drodze i można od niego zacząć poszukiwanie kolejnego najbardziej dowodzącego wierzchołka.

Algorytm 16 Proof-Number Search z wierzchołkiem bieżącym

```

1: procedure PNSEARCH(root, maxNodes)
2:   Evaluate(root)
3:   SetProofAndDisproofNumbers(root)
4:   currentNode ← root
5:   while root.proof ≠ 0 AND root.disproof ≠ 0 AND CountNodes() ≤ maxNodes
6:     do
7:       mostProvingNode ← SelectMostProving(currentNode)
8:       DevelopNode(mostProvingNode)
9:       currentNode ← UpdateAncestors(mostProvingNode)
10:      end while
11:      if root.proof = 0 then
12:        root.value ← true
13:      else if root.disproof = 0 then
14:        root.value ← false
15:      else
16:        root.value ← unknown
17:      end if
18:   end procedure

```

Powyższe spostrzeżenia pozwalają na proste rozwinięcie algorytmu *pn-search* poprzez dodanie dodatkowej zmiennej. Zmienna ta reprezentuje najgłębszy wierzchołek leżący na ścieżce do najbardziej dowodzącego wierzchołka, którego liczby dowodząca i obalająca nie uległy zmianie podczas wykonywania procedury aktualniania. Wierzchołek ten nazywa się wierzchołkiem bieżącym. Uaktualniony algorytm *pn-search* przedstawiono jako alg. 16, a jego rozszerzoną funkcję zwracającą bieżący wierzchołek zaprezentowano jako alg. 17.

Opisane wyżej udoskonalenie redukuje koszt czasowy operacji wyszukiwania najbardziej dowodzącego wierzchołka oraz uaktualniania przodków z wartości liniowej względem głębokości drzewa do wartości bliskiej wartości stałej – co zaobserwowano w eksperymentach z grami *Qubic* i *Go-Moku* [3].

3.5.5. Transpozycje w DAG

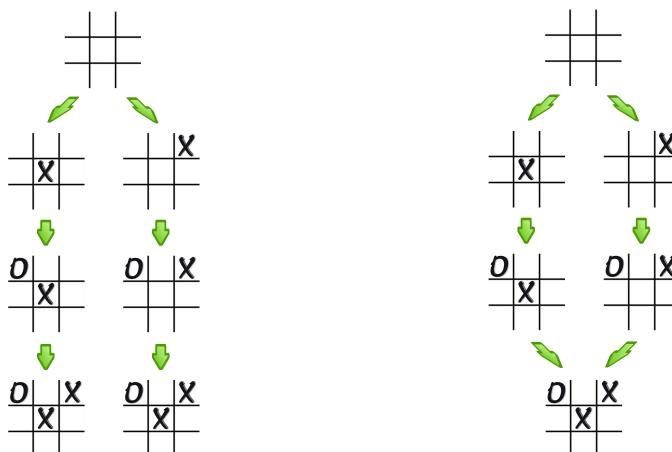
Transpozycje są używane w celu uniknięcia wykonywania kilka razy tych samych obliczeń. Sytuacja taka ma miejsce, jeśli gra pozwala na osiągnięcie pewnego stanu dwiema różnymi drogami. Rozważmy to na przykładzie prostej gry *Kółko i krzyżyk*. Rys. 3.18 obrazuje zaistniałą sytuację.

Algorytm 17 Uaktualniający przodków (rozszerzony)

```

1: function UPDATEANCESTORS(node)
2:   changed  $\leftarrow$  true
3:   while node  $\neq$  NIL AND changed do
4:     oldProof  $\leftarrow$  node.proof
5:     oldDisproof  $\leftarrow$  node.disproof
6:     SetProofAndDisproofNumbers(node)
7:     changed  $\leftarrow$  (oldProof  $\neq$  node.proof OR oldDisproof  $\neq$  node.disproof)
8:     previousNode  $\leftarrow$  node
9:     node  $\leftarrow$  node.parent
10:   end while
11:   return previousNode
12: end function

```



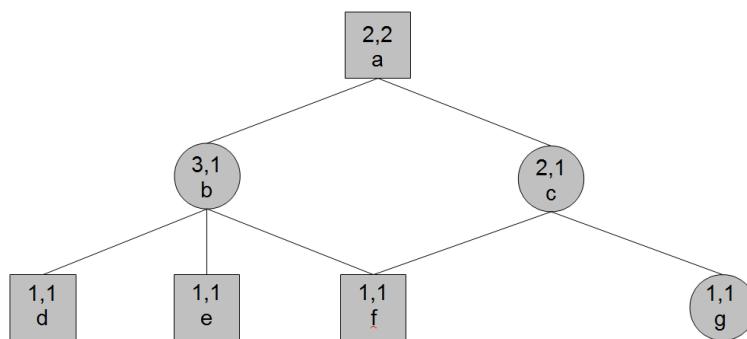
(a) ten sam stan osiągnięty różnymi sek- (b) połączenie redundantnych stanów w
wencjami posunięć jeden

RYSUNEK 3.18. Przykład transpozycji

Mówiąc o transpozycjach wykluczamy z rozważań sytuacje, gdy pewien stan gry może zostać powtórnie osiągnięty w trakcie tej samej rozgrywki. Jeśli w grze każde posunięcie jest przekształcającym czyli takim, po którego wystąpieniu żaden z wcześniejszych stanów nie jest już osiągalny dowolną kombinacją poprawnych posunięć¹ to najbardziej odpowiednią do współpracy z algorytmami *best-first search*

¹Przykładowo w szachach jest to bicie, roszada, ruch pionkiem. Gry, w których każde posunięcie jest przekształcające to: *Pentomino*, *Quarto*, szerzej opisane w sekcji 2.3.

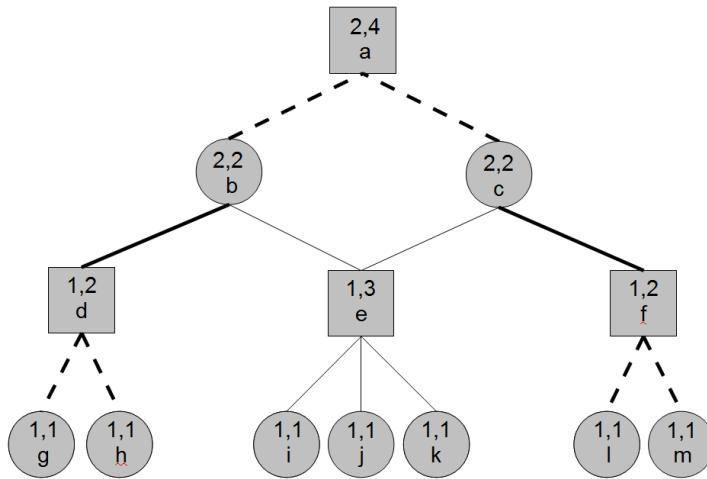
wydaje się struktura digrafu skierowanego, w skrócie DAG, opisanego w sekcji 3.4.2. Pozwala ona łączyć identyczne stany gry w jednym wierzchołku, który będziemy nazywać wierzchołkiem transpozycyjnym, zapobiegając wielokrotnemu wykonywaniu tych samych obliczeń oraz przechowywaniu w pamięci tych samych poddrzew gry. Jednak algorytm *pn-search* w swojej klasycznej wersji nie jest przystosowany do współpracy z DAG i może podczas swojej pracy natrafić na kilka niedogodności. Na rys. 3.19 zilustrowano jeden z problemów na jaki może on napotkać podczas współpracy z DAG.



RYSUNEK 3.19. Dwukrotnie policzony wierzchołek obalający

Wierzchołek *f* jest tworzony dwukrotnie, najpierw podczas rozwijania wierzchołków potomnych wierzchołka *b*, a następnie podczas rozwijania wierzchołka *c*. Liczby dowodząca i obalająca wierzchołków *b* i *c* są bez zarzutu. Jednak wartość obalająca wierzchołka *a* wyliczona według standardowej procedury algorytmu *pn-search* jako suma wartości liczb obalających wszystkich swoich potomków, jest niepoprawna. Należy zauważyć, że wyliczenie wierzchołka *f* do wartości *false* pozwala na przypisanie tych samych wartości wierzchołkom *b* i *c*, a następnie wierzchołkowi *a*. Z tego wynika, że jego liczba obalająca powinna być równa 1. Wierzchołek *f* jest więc uwzględniony dwukrotnie w najmniejszym zbiorze obalającym.

Innym problemem jest sytuacja przedstawiona na rys. 3.20. Tutaj wartość liczby obalającej wierzchołka *a* wynosi 4, ponieważ uwzględnia wierzchołki *g*, *h*, *l*, *m*. Jednak jak łatwo zauważyć, wyliczenie tylko trzech wierzchołków ze zbioru *i*, *j*, *k* pozwala obalić wierzchołek *a*. Wartość liczby obalającej wierzchołka *a* powinna więc wynosić 3. Ponadto błędne wartości to nie jedyny problem na jaki może natrafić algorytm *pn-search*. Założymy, że wartość liczb dowodzącej i obalającej wierzchołka *a* jest poprawna i wynosi odpowiednio 2 i 3. Zobaczmy jak zachowią się funkcja wyszukiwania najbardziej dowodzącego wierzchołka. W pierwszym kroku zostanie wybrany jeden z wierzchołków *b* lub *c* ponieważ ich liczby dowodzące są równe, jednak w kroku



RYSUNEK 3.20. Zignorowany lepszy wierzchołek obalający

następny zostanie wybrany zależnie od poprzedniego wyboru, wierzchołek d lub f . Wierzchołek e nie zostanie wybrany, ponieważ jego liczba obalająca jest większa od liczb obalających wierzchołków d i f , ale to przecież przez niego prowadzi droga do trzech najbardziej obalających wierzchołków i, j, k .

Pomimo tych niedoskonałości wiemy, że w DAG zawsze istnieje najbardziej dowodzący wierzchołek. Martin Schijf zaproponował algorytm, który poprawnie taki wierzchołek znajduje. Niestety jest to algorytm czysto teoretyczny i jego zastosowanie w praktyce ze względu na nieefektywną złożoność obliczeniową jak i pamięciową, jest niemożliwe. Dowód twierdzenia, że w DAG zawsze istnienie najbardziej dowodzący wierzchołek oraz wspomniany algorytm, możemy znaleźć w [54].

W praktyce możemy zastosować standardowy algorytm *pn-search* wprowadzając w nim trzy zmiany:

1. po wyszukaniu najbardziej dowodzącego wierzchołka rozwijane są wszystkie wierzchołki reprezentujące ten sam stan¹. W celu zapobiegania budowy dużych drzew powinny jednak zostać one połączone ze sobą w jednym wierzchołku transpozycyjnym, tworząc w ten sposób DAG;
2. procedura uaktualniająca przodków powinna uaktualniać wszystkich, a nie jak dotychczas tylko jednego rodzica.

¹Taka sytuacja ma miejsce jeśli nie łączymy transpozycji w jeden wierzchołek.

3. podczas dodawania nowego wierzchołka należy sprawdzać czy nie został on już wcześniej utworzony w grafie, a jeśli tak, to dodajemy tylko nową krawędź pomiędzy rozwijanym a istniejącym wierzchołkiem transpozycyjnym. Procedurę rozwijającą w ten sposób wierzchołek przedstawiono jako alg. 18.

Algorytm 18 Rozwijający wierzchołek z uwzględnieniem transpozycji (*wersja z natychmiastowym wyliczaniem*)

```

1: procedure DEVELOPNODE(node)
2:   GenerateAllChildren(node)
3:   for all child in node.children do
4:     if IsAlreadyInDAG(child) then
5:       MakeEdge(node, child)
6:     else
7:       Evaluate(child)
8:       SetProofAndDisproofNumbers(child)
9:     end if
10:    end for
11: end procedure
```

Należy zauważyć, iż zmieni się interpretacja liczb dowodzącej i obalającej i będą one teraz górnym oszacowaniem na moc odpowiadających im minimalnych zbiorów w DAG. Dla omawianego algorytmu można zastosować kilka wymienionych niżej udoskonaleń:

- aby wyszukiwanie ewentualnych powtórzeń nowo dodawanych wierzchołków było efektywne, należy do przechowywania DAG posłużyć się strukturami danych opartymi o drzewa zrównoważone, przykładowo *AVL* lub *czerwono-czarne*, pozwoli to na wyszukiwanie wierzchołka w czasie $O(\log n)$;
- jako wierzchołek transpozycyjny można również traktować wierzchołek symetryczny do już istniejącego. Pozwala to na skutecną redukcję liczby wierzchołków w grach o dużej liczbie niezależnych osi symetrii;
- opisane w sekcji 3.5.4 usprawnienie przerywające aktualnianie przodków, jeśli żadna z liczb nie uległa zmianie, można zastosować również w tym przypadku. W grach, w których dla dowolnego wierzchołka długość każdej drogi do korzenia jest równa, przykładowo *Czwórki*, *Qubic*, *Go-Moku*, zamiast odkładać

wierzchołki na stos podczas uaktualniania rodziców, możemy zastosować kolejkę. Pozwoli to na uniknięcie wielokrotnego uaktualniania wierzchołków zawierających się w więcej niż jednej ścieżce;

- usuwanie poddrzew, opisane w sekcji 3.5.2, nie jest już takie proste w przypadku DAG. Trzeba uważać czy poddrzewo rozwiązanego wierzchołka nie zawiera wierzchołków należących do innego poddrzewa, jakiegoś nie wyliczonego jeszcze wierzchołka. W drzewach AND/OR każde poddrzewo było podczepione tylko pod jeden wierzchołek, w DAG sytuacja niestety nie jest już taka klarowna. Dodatkowo może się okazać, że nowo rozwijany wierzchołek był transpozycją któregoś z wierzchołków wcześniej usuniętego poddrzewa. Może to skutkować wielokrotnym budowaniem identycznych poddrzew. Jedno jest pewne – można usunąć wierzchołki końcowe, opisane przez instrukcję *Evaluate(node)* jako *true* lub *false*. W rozwiązanych wierzchołkach wewnętrznych można również usunąć wszystkie graniczne wierzchołki potomne, które nie są transpozycyjne.

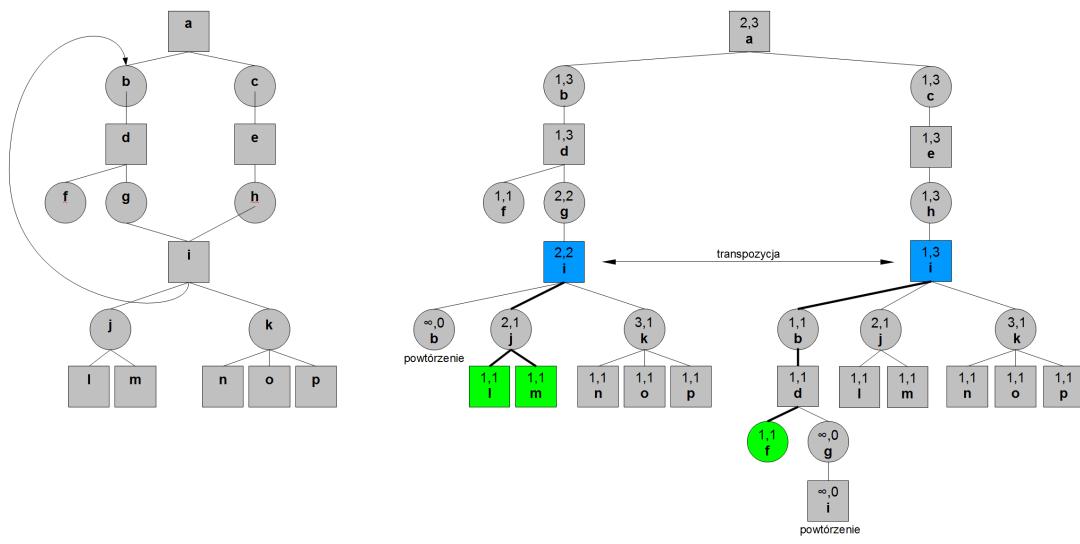
Opisany algorytm był z powodzeniem stosowany dla gier: *Czwórki* [2], *Qubic*, *Go-Moku* [3].

3.5.6. Transpozycje w DCG

Jeśli z dowolnego stanu gry można po pewnej liczbie posunięć osiągnąć ten sam stan to taką sytuację będziemy nazywać powtórzeniem. Powtórzenia występują w wielu znanych grach. Dla przykładu są to: *Szachy*, *Warcaby* oraz wiele wariantów *Go*. Najbardziej naturalną strukturą dla zobrazowania takiej gry jest DCG (directed cyclic graph) - digraf cykliczny opisany w sekcji 3.4.3. Liczba powtórzeń jaka może wystąpić w grze zależy od jej zasad. W *Młynku* już pierwsze powtórzenie stanu gry oznacza remis. W *Awari* bądź w *Szachach* remisowe jest dopiero drugie powtórzenie (trzykrotne wystąpienie tego samego stanu gry). Istnieje również duża liczba gier, dla których nie ma żadnych zasad związanych z występowaniem powtórzeń. Jeżeli gracze doprowadzą do sytuacji, w której żaden z nich poprzez pewną sekwencję posunięć (niezależnie od posunięć przeciwnika) nie może doprowadzić do własnej wygranej i obydwa będą przedłużać rozgrywkę w nieskończoność, poprzez pewną powtarzaną sekwencję ruchów, to taka sytuacja również jest traktowana jako remisowa. Mimo to liczba powtórzeń z punktu widzenia technik przeszukujących nie ma znaczenia i już na pierwszym z nich możemy zakończyć poszukiwania, traktując je jako pozycję końcową. A to dlatego, że poddrzewa podczepione pod pierwsze, jak i pod kolejne wystąpienia powtórzonego wierzchołka, są identyczne. Następnie

założymy, że to drugie wystąpienie traktowane jest jako pozycja końcowa zakończona remisem i tak je oznaczamy¹. Teraz mogą wystąpić dwie możliwości:

1. wierzchołek zawierający pierwsze wystąpienie powtórzonej pozycji również zostanie wyliczony jako remisowy – wtedy obydwa¹ wyliczenia są poprawne;
2. wierzchołek zawierający pierwsze wystąpienie powtórzonej pozycji nie zostanie wyliczony jako remisowy – oznacza to, że istnieje optymalna strategia jednego z graczy, prowadząca do jego wygranej i nie może ona zawierać powtórzenia.



(a) digraf cykliczny DCG

(b) drzewo powstałe po konwersji

RYSUNEK 3.21. Konwersja DCG do drzewa

Największym problemem w DCG jest fakt, że niemożliwe jest przypisanie jednoznacznych wartości liczb dowodzących i obalających każdemu wierzchołkowi. Jest to związane z różną historią związaną z wierzchołkami reprezentującymi ten sam stan gry. Pod pojęciem „historią wierzchołka” należy rozumieć kolejne stany na ścieżce, poczynając od korzenia, które zaistniały w grze przed osiągnięciem rozważanego wierzchołka. Problem ten nazywany jest w literaturze „problemem grafu historii interakcji” (ang. *GHI*² - *problem*) [11, 12, 26, 29]. W bieżącej sekcji omówimy związane z nim trudności na przykładach z [54].

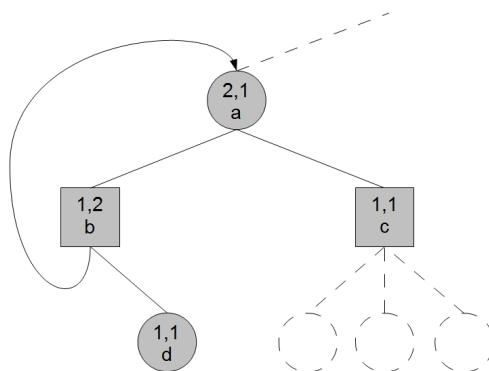
¹Pamiętamy, że wartość remisową możemy reprezentować tylko jako jedną z wartości zbioru: *true*, *false*.

²Poprawne dla pierwszego wystąpienia i pierwszego powtórzenia, które jest drugim wystąpieniem identycznego stanu.

²Akronim od Graph History Interaction.

Na rys. 3.21 zobrazowany jest ten problem. Rys. 3.21(a) przedstawia DCG, który został zamieniony na drzewo, przedstawione na rys. 3.21(b). Drzewo to zawiera ścieżki, jakie występują we wspomnianym grafie. Ten sam wierzchołek i zależnie od ścieżki, którą został osiągnięty, zawiera różne wartości liczby dowodzącej i obalającej. Jeżeli wybierzemy ścieżkę $(a - b - d - g - i)$ to minimalny zbiór dowodzący dla i będzie się składał z wierzchołków $\{l, m\}$, natomiast w przypadku wyboru ścieżki $(a - c - e - h - i)$ do minimalnego zbioru dowodzącego będzie należał jedynie wierzchołek $\{f\}$. Zatem liczba dowodząca wierzchołka i będzie raz równa 2 (udowodnienie wierzchołków: l i m), a raz równa 1 (udowodnienie wierzchołka f). Podobnie ma się sytuacja z liczbami obalającymi.

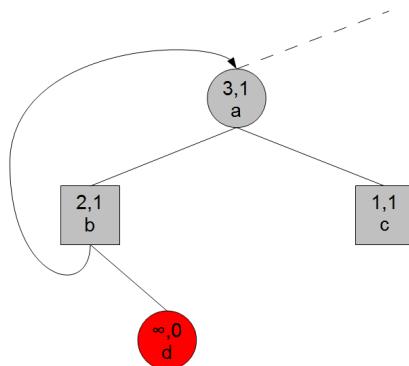
Kolejny problem, na który może napotkać algorytm *pn-search* w DCG jest wywołany istnieniem cykli. Na rys. 3.22 przedstawiono fragment grafu z wyliczonymi liczbami dowodzącymi i obalającymi. Wartości te zostały wyliczone wprost z wartości liczb obalających i dowodzących swoich wierzchołków potomnych.



RYSUNEK 3.22. Problem nieskończonej aktualizacji w DCG

Założymy, że wierzchołek c został rozwinięty generując trzy wierzchołki potomne. Zostały one zaznaczone na rys. przerywaną linią. Nową wartością jego liczby obalającej zostanie 3, przez co stanie się ona większa od wartości liczby obalającej wierzchołka b , a to spowoduje aktualizację liczby dowodzącej w węźle a do wartości 2. Po czym liczba obalająca wierzchołka b stanie się nieaktualna i zostanie jej przypisana wartość 3. To spowoduje kolejną aktualizację liczby obalającej wierzchołka a do wartości 3. Następnie znowu wierzchołek b dostanie nową wartość 4 liczby obalającej, co wymusi kolejną aktualizację wierzchołka a , ale nie wprowadzającej już jakiekolwiek zmiany. Widzimy zatem, że wartość wierzchołka a została zaktualizowana aż trzy razy. W skrajnym przypadku kiedy wierzchołek c zostanie wyliczony do *false* i jego liczba obalająca wyniesie ∞ spowoduje to nieskończoną liczbę aktualizacji.

Wydawać by się mogło, że wymuszenie co najwyżej pojedynczej aktualizacji dla każdego wierzchołka podczas operacji rozwijania wierzchołków potomnych, mogłoby rozwiązać problem.



RYSUNEK 3.23. Problem opóźnionego wyliczenia wartości wierzchołka w DCG

Rozważmy jednak ponownie sytuację przedstawioną na rys. 3.22. Założymy, że wierzchołek c jeszcze nie został rozwinięty, natomiast wierzchołek d został wyliczony do $false$. Sytuacja taka została zobrazowana na rys. 3.23. Ponieważ wartość liczby obalającej wierzchołka a jest równa sumie tych liczb w b i c , to w a zawsze będzie ona większa o jeden niż w b . Natomiast wartość liczby obalającej wierzchołka b jest równa mniejszej z tych liczb w a i d , czyli zawsze będzie równa tej z a . To powoduje, że po nieskończonej liczbie iteracji wartość a powinna wynieść ∞ . Ponieważ założyliśmy, że każdy wierzchołek nie może zostać uaktualniony więcej niż jeden raz, wartość wierzchołka a wyniesie $(3,1)$. Kolejnym najbardziej dowodzącym wierzchołkiem musi zostać c , ponieważ jest to jedyny graniczny nie rozwiązany wierzchołek, w związku z czym może nastąpić jeszcze wiele niepotrzebnych obliczeń, zanim zostanie ustalona wartość wierzchołka a .

Te dwa wymienione przykłady sugerują nam, abyśmy unikali cykli w praktycznych zastosowaniach. W [54] proponuje się, aby dokonać tego na trzy sposoby:

1. zignorowanie wszystkich rodzajów transpozycji poprzez zastosowanie struktury drzewa. W praktyce może być to użyteczne tylko dla gier, w których stany gry powtarzają się rzadko. Inaczej większość pamięci zostanie zmarnowana do przechowywania i obliczania identycznych wierzchołków, a w rezultacie zasoby pamięciowe mogą się bardzo szybko wyczerpać;
2. wybieranie na wierzchołki transpozycyjne tylko tych, które wystąpiły bezpośrednio po posunięciu przekształcającym, omówionym szczegółowo w sekcji 3.5.5.

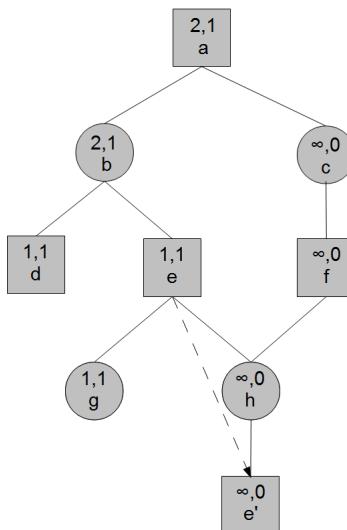
Wynika to z faktu, że w poddrzewie podczepionym do wierzchołka a reprezentującego stan po posunięciu przekształcającym, nie może znaleźć się żaden z wierzchołków występujących na ścieżce od korzenia do a . Zatem ewentualna transpozycja może zostać utworzona tylko z wierzchołkiem, który również reprezentuje stan po przynajmniej jednym posunięciu przekształcającym, a to wyklucza możliwość powstania cyklu. W tym przypadku niestety wciąż są wielokrotnie rozwijane wierzchołki, przechowujące identyczne stany gry, co nadal skutkuje nieoptymalnym wykorzystaniem pamięci;

3. ograniczenie do dwóch liczb wierzchołków zawierających identyczne pozycje. Jeden z nich reprezentuje wszystkie pierwsze wystąpienia danego stanu w grafie, natomiast drugi reprezentuje wszystkie pierwsze powtórzenia tego stanu. Wynika z tego, że obydwa te wierzchołki mogą posiadać większą od jedynki liczbę wierzchołków rodzicielskich. Wymaga ono sprawdzenia, podczas dodawania do DAG nowego wierzchołka a' , czy nie istnieje wierzchołek a reprezentujący identyczny stan gry. Jeśli okaże się, że wierzchołek a znajduje się w DAG, to mogą nastąpić dwa przypadki:
 - **wierzchołek a nie znajduje się na najkrótszej drodze pomiędzy wierzchołkiem a' i korzeniem** – nowo generowana pozycja a' jest zatem transpozycją i zostanie z a połączona w jeden wierzchołek transpozycyjny;
 - **wierzchołek a znajduje się na najkrótszej drodze pomiędzy wierzchołkiem a' i korzeniem** – pozycja a' traktowana jest zatem jako powtórzenie i w celu uniknięcia cyklu zostanie oznaczona jako wierzchołek końcowy. Liczby dowodząca i obalająca zostaną zainicjalizowane zgodnie z przyjętą interpretacją pozycji remisowej.

Załóżmy teraz, że wystąpił drugi przypadek, a remis jest interpretowany jako wartość *false*. Powtórnemu wierzchołkowi a' zostaje więc przypisana wartość *false* a zgodnie z nią para liczb $(\infty, 0)$. Jeśli wierzchołek a reprezentujący pierwsze wystąpienie zostanie rozwiązany do wartości *false*, wartość powtózonego wierzchołka a pozostaje bez zmian. W przypadku kiedy wartość wierzchołka a zostanie wyliczona do *true*, również wartość wierzchołka a' musi zostać zamieniona na *true*, a wartość liczb dowodzącej i obalającej zamieniona na $(0, \infty)$. Wymusi to aktualizację wszystkich jego przodków. I tutaj napotykamy na kolejne zagrożenie a to dlatego, że jednym z przodków a' jest wierzchołek a , który wywołał aktualizację a' . Może zatem powstać nieskończona pętla aktu-

alizacji. Najlepszym rozwiązaniem jest zastosowanie wcześniej przedstawionej rozszerzonej procedury uaktualniania przodków (alg. 17), przerywającej proces ich uaktualniania w przypadku braku zmiany kolejkowej z liczb: dowodzącej bądź obalającej jakiegokolwiek wierzchołka rodzicielskiego.

Omawiany trzeci wariant zapewnia, że jeśli korzeniowi zostanie przypisana wartość *true* to na pewno jest ona poprawna. Niestety jeżeli korzeń otrzyma wartość *false* musimy liczyć się z faktem, że pewien dowodzący wariant mógł zostać pominięty. Taki pesymistyczny przypadek przedstawiono na rys. 3.24. Wierzchołek e' reprezen-



RYSUNEK 3.24. Przeoczona wygrana

tuje ten sam stan co wierzchołek e . Założymy, że wierzchołek d zostaje obalony. Powoduje to również obalenie wierzchołka b , a następnie a , co w rezultacie prowadzi do obalenia całego drzewa. Zauważmy jednak, że algorytm wykonywałby się dalej i w przyszłości wierzchołek g zostałby udowodniony. To spowodowałoby udowodnienie e , co wywołuje z kolei opisywaną wcześniej aktualizację wartości e' do *true* i ciągu kolejnych aktualizacji, udowadniających wierzchołki na drodze: $h - f - c - a$. W rezultacie drzewo zostaje udowodnione. Niestety taka sytuacja zostanie przegapiona, jeżeli wcześniej omawiany wierzchołek d będzie posiadał wartość *false*. Ten trzeci sposób unikania cykli to tak naprawdę wcześniej opisany algorytm dla DAG. Z tą różnicą, że procedura rozwijająca wierzchołek oprócz uwzględniania transpozycji powinna dodatkowo uwzględniać powtórzenia (alg. 19). Również procedura uaktualniająca przodków, przedstawiona jako alg. 14, powinna w przypadku przypisywania wierzchołkowi wartości *true*, uaktualniać wszystkie powtórzenia tego wierzchołka, oczywiście jeśli takie istnieją. Wymaga to przechowywania listy wskaźników

Algorytm 19 Rozwijający wierzchołek z uwzględnieniem transpozycji i powtórzeń (*wersja z natychmiastowym wyliczaniem*)

```

1: procedure DEVELOPNODE(node)
2:   GenerateAllChildren(node)
3:   for all child in node.children do
4:     if IsAlreadyInDAG(child) then
5:       if IsRepetition(child) then
6:         child.value  $\leftarrow$  false
7:         child.evaluate  $\leftarrow$  true
8:         SetProofAndDisproofNumbers(child)
9:         node.repetition  $\leftarrow$  child
10:      else
11:        MakeEdge(node, child)
12:      end if
13:    else
14:      Evaluate(child)
15:      SetProofAndDisproofNumbers(child)
16:    end if
17:   end for
18: end procedure

```

do wszystkich powtórzeń każdego wierzchołka. A dany wskaźnik powinien być dodawany podczas rozwijania powtózonego wierzchołka (linia 9 alg. 19). Po takiej aktualizacji wartości wierzchołka powtózonego należy wykorzystać rozszerzoną procedurę uaktualniania przodków, opisaną jako alg. 17, ponieważ w przypadku zastosowania standardowej procedury, wierzchołki zaczynają się – jak już wspomniano – uaktualniać w nieskończonej pętli.

3.5.7. Algorytm BTA Proof-Number Search

Algorytm BTA jest rozwiążaniem pozwalającym uniknąć przeoczenia wygranej, które może mieć miejsce podczas współpracy algorytmu *pn-search* ze strukturą DCG, znanego jako *GHI-problem*. Problem ten został szczegółowo omówiony w sekcji 3.5.6.

W celu rozróżnienia pomiędzy wierzchołkami reprezentującymi identyczny stan, a różniącymi się historią, wprowadza się podział na dwa nowe rodzaje wierzchołków: *bazowe* i *bliźniacze*. Jeżeli podczas dodawania nowego wierzchołka w grafie istnieje już identyczny to dodawany wierzchołek oznaczamy jako bliźniaczy. W przeciwnym

przypadku wierzchołek jest wierzchołkiem bazowym. Każdy wierzchołek bliźniaczy zawiera wskaźnik na odpowiadający mu wierzchołek bazowy¹. Wierzchołki bazowe nie zawierają wskaźników na swoich „bliźniaków”. Aby zapobiec wielokrotnemu rozwijaniu identycznych poddrzew, wierzchołki bliźniacze pozostają nie rozwinięte, jednak ich historia, a z niej wynikająca ich wartość, może się różnić od odpowiadających im wierzchołków bazowych. Dodatkowo w każdym wierzchołku (zarówno bazowym jak i bliźniaczym) przechowujemy teraz trzy dodatkowe informacje:

- głębokość wierzchołka w drzewie;
- znacznik – *prawdopodobny remis*;
- głębokość na jakiej remis może wystąpić – *pdDepth*.

Te trzy zmienne pozwalają na poprawne rozpoznawanie rzeczywistych sytuacji remisowych. Ponieważ w tym przypadku powtórzenie pozycji nie może być interpretowane jako remis, w pierwszym kroku (podczas wykonywania procedury *Select-MostProoving(node)*) powtórzone wierzchołki oznaczamy jako *prawdopodobny remis*, a głębokość na jakiej znajduje się (na aktualnej ścieżce poszukiwań) odpowiadający im wierzchołek reprezentujący pierwsze wystąpienie, jest zapamiętywana w zmiennej *pdDepth* powtórnego wierzchołka. Wartość *pdDepth* jest interpretowana jako głębokość na jakiej może wystąpić sytuacja remisowa.

Główna część algorytmu BTA została przedstawiona jako alg. 20. Widać, że różnice w porównaniu do standardowego *pn-search* (alg. 9) są niewielkie. Pierwszą jest inicjacja (w linii nr. 4) głębokości korzenia na 0, a w procedurze *UpdateAncestors(node)* (znajdującej się w linii nr. 9) miejsce dawniejszego parametru czyli najbardziej dowodzącego wierzchołka, zajmuje jego rodzic. To dlatego, że teraz wartość najbardziej dowodzącego wierzchołka jest już wyliczana podczas wykonywania procedury *DevelopNode(node)* w linii nr. 7.

Największe zmiany zostały wprowadzone w funkcji *SelectMostProoving(node)* i procedurach: *DevelopNode(node)* oraz *UpdateAncestors(node)*. Procedury *Evaluate(node)* oraz funkcja *CountNodes()* nie uległy zmianie. Natomiast procedura *SetProofAndDisproofNumbers(node)* podczas aktualizacji wartości liczb dowodzącej i obalającej wierzchołka bliźniaczego, korzysta z odpowiednich wartości wierzchołków potomnych wierzchołka bazowego, odpowiadającego aktualizowanemu, bliźniaczemu wierzchołkowi.

¹Realizacja tych zmian wymaga odpowiedniej modyfikacji funkcji *GenerateAllChildren(node)*.

Algorytm 20 BTA Proof-Number Search

```

1: procedure PNSEARCH(root, maxNodes)
2:   Evaluate(root)
3:   SetProofAndDisproofNumbers(root)
4:   root.depth ← 0
5:   while root.proof ≠ 0 AND root.disproof ≠ 0 AND CountNodes() ≤ maxNodes
6:     do
7:       mostProvingNode ← SelectMostProving(root)
8:       DevelopNode(mostProvingNode)
9:       mostProvingNode ← ParentOnCurrentSearchPath(mostProvingNode)
10:      UpdateAncestors(mostProvingNode, root)
11:    end while
12:    if root.proof = 0 then
13:      root.value ← true
14:    else if root.disproof = 0 then
15:      root.value ← false
16:    else
17:      root.value ← unknown
18:    end if
19: end procedure

```

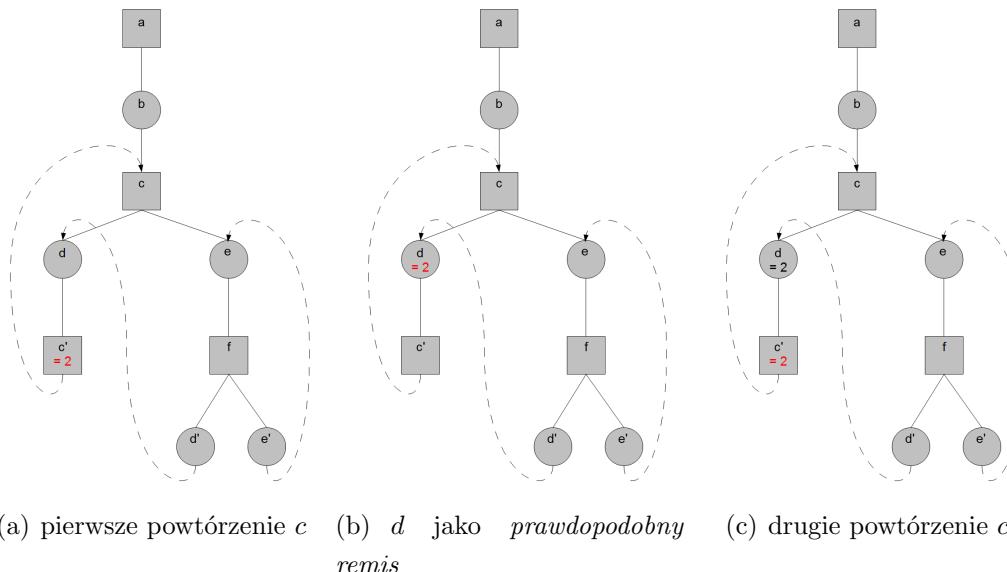
Analogicznie jak w klasycznym algorytmie *pn-search* najważniejsza jego część odbywa się w pętli pomiędzy liniami: 5 – 10, w której wykonywane są kolejno trzy instrukcje: wyszukiwanie najbardziej dowodzącego wierzchołka, jego rozwinięcie oraz uaktualnianie wartości jego przodków.

Funkcja wyszukująca najbardziej dowodzący wierzchołek rozpoczęta poszukiwania od korzenia i może natrafić na dwa przypadki:

1. istnieje jeden wierzchołek potomny analizowanego wierzchołka, który jest oznaczony jako *prawdopodobny remis*, a pozostałe z nich są oznaczone jako *prawdopodobny remis*, albo posiadają wartość *false*. Wówczas analizowany wierzchołek zostaje oznaczony jako *prawdopodobny remis*, a wartość zmiennej *pdDepth* zostaje wybrana jako minimum z wartości *pdDepth* jego wierzchołków potomnych posiadających etykietę *prawdopodobny remis*. W kolejnym kroku znaczniki *prawdopodobny remis* są usuwane ze wszystkich wierzchołków potomnych aktualizowanego wierzchołka, a poszukiwanie wierzchołka najbardziej dowodzącego zostaje rozpoczęte od jego rodzica;

2. w każdym innym przypadku najbardziej dowodzący wierzchołek jest wybierany według klasycznej procedury z pominięciem wierzchołków oznaczonych jako *prawdopodobny remis*.

Załóżmy, że pewien wierzchołek w przeszukiwanej ścieżce na głębokości d jest oznaczony jako *prawdopodobny remis*, a wartość jego zmiennej $pdDepth$ równa jest głębokości d . Oznacza to, że znacznik *prawdopodobny remis* w tym wierzchołku ustalił się tylko w oparciu o powtórzenia tego wierzchołka bądź rzeczywiste pozycje remisowe występujące w jego poddrzewie. W takim wypadku wierzchołek zostaje oznaczony jako *rzeczywisty remis*¹ – niezależny od historii tego wierzchołka.



RYSUNEK 3.25. Kroki 1-3 wykonywania funkcji $SelectMostProving(node)$

Aby lepiej zrozumieć tę sytuację posłużymy się przykładem zaczerpniętym z [11]. Na rys. 3.25(a) przedstawiono część grafu. Funkcja wyszukiwania najbardziej dowodzącego wierzchołka $SelectMostProving(node)$ rozpoczęła poszukiwanie od korzenia (wierzchołek a) i dociera do wierzchołka c' , który jest powtórzeniem² wierzchołka c . Zostaje on oznaczony jako *prawdopodobny remis*³, a jego zmiennej $pdDepth$ ⁴ zostaje przyp-

¹Rzeczywisty remis oznaczamy inicjując wartością liczbą dowodzącą i obalającą identycznie jak dla wartości *false*.

²Zakładamy, że wierzchołki przeszukiwane są w kolejności od lewej do prawej strony.

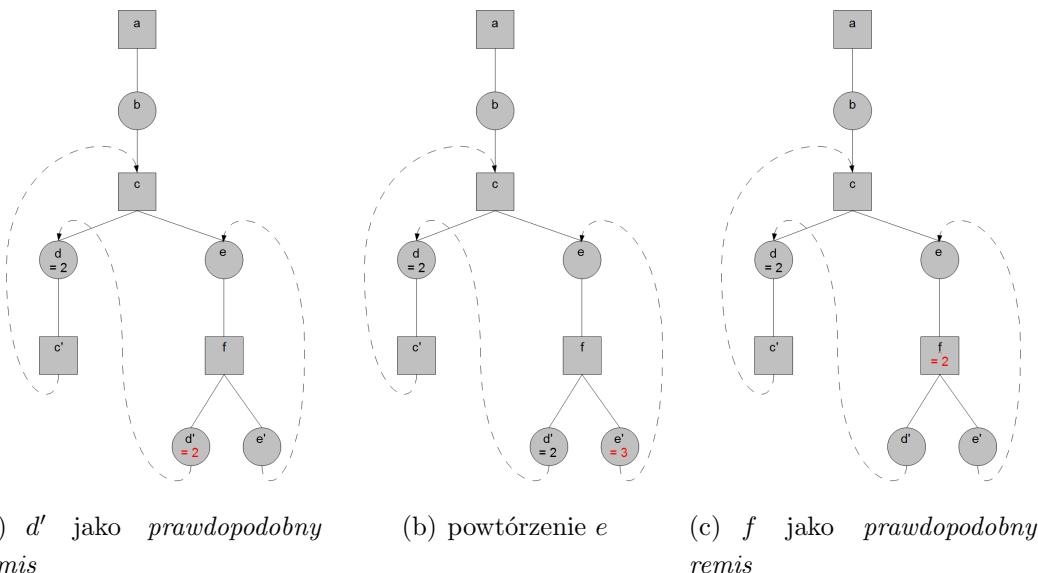
³Zostało to przedstawione za pomocą znaku równości.

⁴Wartość zmiennej $pdDepth$ znajduje się po prawej stronie znaku równości.

isana wartość 2, ponieważ jest ona głębokością na jakiej znajduje się wierzchołek c .

Następnym wierzchołkiem, od którego wznawiamy poszukiwanie jest wierzchołek d . Posiada on tylko jeden wierzchołek potomny, który jest oznaczony jako *prawdopodobny remis*, a zatem również on zostaje tak oznaczony. Wartość jego zmiennej $pdDepth$ zostaje zainicjowana najmniejszą analogiczną wartością spośród wszystkich jego wierzchołków potomnych (w tym przypadku tylko jeden) oznaczonych jako *prawdopodobny remis*. Następnie znacznik *prawdopodobny remis* wierzchołka c' jest czyszczony¹. Teraz sytuacja wygląda jak na rys. 3.25(b).

Analogicznie następnym wierzchołkiem, od którego kontynuujemy poszukiwania jest c (rodzic d). Jedynym jego wierzchołkiem potomnym nie oznaczonym jako *prawdopodobny remis* jest e . Kontynuując poszukiwania znowu natrafiamy (tym razem poprzez kolejne wierzchołki na drodze: $c - e - f - d' - c'$)² na powtórzenie wierzchołka c . Ponownie wierzchołek c' zostaje oznaczony jako *prawdopodobny remis*, a jego zmiennej znowu zostaje przypisana wartość 2. Sytuacja jest przedstawiona na rys. 3.25(c).



RYSUNEK 3.26. Kroki 4-6 wykonywania funkcji *SelectMostProving(node)*

Tym razem podczas wznawiania poszukiwań rodzicem wierzchołka c' na aktu-

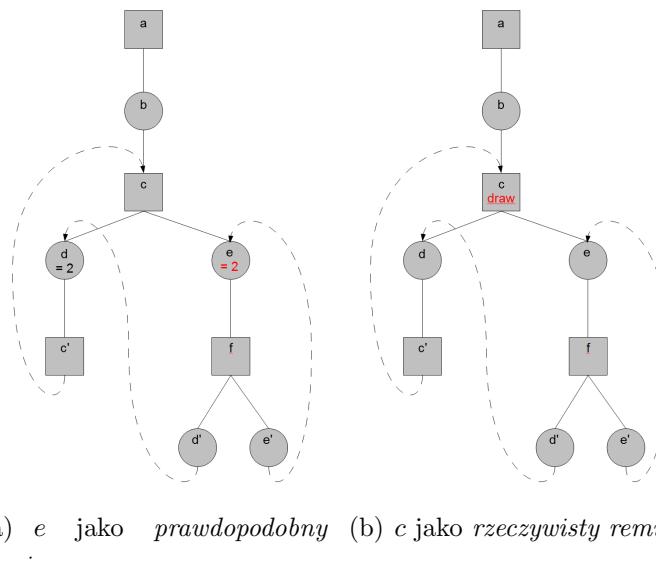
¹Również wartość zmiennej $pdDepth$ przestaje mieć znaczenie.

²Wierzchołek potomny c' zostaje wyszukany poprzez bezpośrednie przekierowanie z d' do jego wierzchołka bazowego.

alnej ścieżce jest d' . Ponieważ wszystkie jego węzły potomne (w tym przypadku tylko jeden) posiadają etykietę *prawdopodobny remis*, zostaje oznaczony jako *prawdopodobny remis*, a znacznik *prawdopodobny remis* wierzchołka c' zostaje wyczyszczony (rys. 3.26(a)).

Poszukiwanie wznowiane jest od rodzica d' czyli wierzchołka f . Jego jedynym węzłem potomnym który nie posiada etykiety *prawdopodobny remis* jest e' , który zarazem okazuje się powtórzeniem wierzchołka e i zostaje oznaczony jako *prawdopodobny remis*, a jego zmienna $pdDepth$ inicjowana jest głębokością wierzchołka e równą 3 (rys. 3.26(b)).

W kolejnym wznowieniu procedury *SelectMostProving(node)* rodzicem e' jest f , a ponieważ wszystkie jego wierzchołki potomne oznaczone są jako *prawdopodobny remis*, również on przyjmuje takie oznaczenie. Najmniejsza wartość $pdDepth$ wśród jego potomków to 2 i taka zostaje przypisana do jego zmiennej. Następnie wartości $pdDepth$ jego wierzchołków potomnych zostają wyczyszczone (rys. 3.26(c)).



RYSUNEK 3.27. Kroki 7-8 wykonywania funkcji *SelectMostProving(node)*

Po oznaczeniu f jego rodzic e również zostaje oznaczony jako *prawdopodobny remis* z $pdDepth = 2$, a znacznik prawdopodobny remis f zostaje wyczyszczony (rys. 3.27(a)).

Następnym wierzchołkiem od którego zaczynamy poszukiwania jest c . Ponieważ wszystkie jego wierzchołki potomne są poetykietowane jako *prawdopodobny remis*, również on otrzymuje etykietę *prawdopodobny remis*, a wartość jego zmiennej

$pdDepth$ to minimum z analogicznych wartości obydwu wierzchołków potomnych, czyli 2. Zauważmy, że jej wartość równa jest głębokości na jakiej znajduje się wierzchołek c . Oznacza to, że wszystkie ścieżki z wierzchołka c prowadzą do powtórzeń zawierających się w poddrzewie tego wierzchołka. Zatem zostaje on oznaczony jako *rzeczywisty remis* (rys. 3.27(b)).

Algorytm 21 BTA wybierający najbardziej dowodzący wierzchołek

```

1: function SELECTMOSTPROVING(node)
2:   if node.twinNode then
3:     baseNode ← node.baseNode
4:   else
5:     baseNode ← node
6:   end if
7:   if baseNode.proof = 0 OR base.Node.disproof = 0 then
8:     return node
9:   else if Repetition(node) then
10:    node.possibleDraw ← true
11:    node.pdDepth ← FindEqualAncestorNode(node).depth
12:    node ← ParentOnCurrentSearchPath(node)
13:    return SelectMostProving(node)
14:   else if NOT baseNode.expanded then
15:     return node
16:   
```

▷ cdn.

Omawianą funkcję wyszukującą najbardziej dowodzący wierzchołek przedstawiono jako alg. 23 i 22. Zaczyna ona poszukiwanie od korzenia. Najbardziej dowodzący wierzchołek jest wyszukiwany w zależności od spełnionych następujących kryteriów:

- **jeżeli wierzchołek bazowy analizowanego wierzchołka jest rozwiązyany** (linia nr. 7) – to jest on zwracany w celu aktualizacji grafu o tę informację. Sytuacja taka może mieć miejsce, ponieważ wierzchołki bazowe nie posiadają wskaźników na swoich bliźniaków. Zatem rozwiązanie wierzchołka bazowego może spowodować, że informacje ich wierzchołków bliźniaczych będą nieaktualne. Aktualizacja zostaje dokonana poprzez następną w kolejności procedurę – *DevelopNode(mostProvingNode)*;
- **jeżeli wierzchołek jest powtórzeniem** (linia nr. 9) – to oznaczamy go jako *prawdopodobny remis*, a zmienna $pdDepth$ reprezentująca głębokość możliwej

pozycji remisowej inicjowana jest głębokością na jakiej znajduje się rodzic odpowiadającego mu wierzchołka bazowego. Dalsze badanie powtórzonego wierzchołka nie jest już potrzebne i szukanie jest wznowiane poczynając od jego rodzica. Funkcja *Repetition(node)* sprawdza zatem, czy na bieżącej ścieżce poszukiwań prowadzącej od korzenia do pozycji węzła *node* ta pozycja wystąpiła;

- **jeżeli wierzchołek nie jest powtóreniem i nie został rozwinięty** (linia nr. 14) – to jest wierzchołkiem granicznym, a zarazem poszukiwanym najbardziej dowodzącym wierzchołkiem. Zostaje więc zwrócony przez funkcję.
- **w każdym innym przypadku** (linia nr. 17) – funkcja *SelectBestChild(node, baseNode, OUT pdPresent)* wywoływana w linii nr. 18 (jej działanie zostanie omówione później) wyszukuje wierzchołek potomny, od którego wznowiane jest poszukiwanie najbardziej dowodzącego wierzchołka.

Jeżeli funkcja ta nie zwróci żadnego wierzchołka oznacza to, że albo każdy z potomków jest rozwiązany (udowodniony w przypadku wierzchołka AND, obalony w przypadku wierzchołka OR), albo każdy jest oznaczony jako *prawdopodobny remis*.

W pierwszym przypadku wierzchołek bazowy analizowanego wierzchołka otrzymuje wartość któregoś z jego wierzchołków potomnych¹. Zostaje on rozwiązany, a analizowany wierzchołek zostaje zwrócony przez funkcję. Jego wartość zostanie zaktualizowana (zostanie rozwiązany) poprzez następną w kolejności procedurę – *DevelopNode(mostProvingNode)*.

W drugim przypadku każdy z wierzchołków potomnych był oznaczony jako *prawdopodobny remis*, więc ich rodzic (analizowany wierzchołek) również zostaje oznaczony jako *prawdopodobny remis*, a wartość jego zmiennej *pdDepth* zostaje zainicjowana minimalną wartością *pdDepth* z wszystkich jego dzieci. Następnie wszystkim wierzchołkom potomnym zostaje usunięty znacznik *prawdopodobny remis*, a poszukiwanie najbardziej dowodzącego wierzchołka jest wznowiane począwszy od rodzica analizowanego wierzchołka.

Zadaniem funkcji *SelectBestChild(node, baseNode, OUT pdPresent)* (alg. 23) jest wybranie najlepszego wierzchołka potomnego. Jest ona wywoływana wewnątrz funkcji *SelectMostProving(node)* i otrzymuje trzy argumenty:

- ***node*** – rodzic, którego najlepsze dziecko będzie wybierane;

¹ W alg. 22 jest to wartość pierwszego wierzchołka, linie: 38 – 39.

Algorytm 22 BTA wybierający najbardziej dowodzący wierzchołek cd.

```

17:   else
18:     bestChild ← SelectBestChild(node, baseNode, pdPresent)
19:     if bestChild = NIL then
20:       if pdPresent then
21:         node.possibleDraw ← true
22:         node.pdDepth ← ∞
23:         for all child in baseNode.children do
24:           if child.possibleDraw then
25:             if child.pdDepth < node.pdDepth then
26:               node.pdDepth ← child.pdDepth
27:               child.possibleDraw ← false
28:             end if
29:           end if
30:         end for
31:         if node.depth = node.pdDepth then
32:           return node
33:         else
34:           node ← ParentOnCurrentSearchPath(node)
35:           return SelectMostProving(node)
36:         end if
37:       else
38:         baseNode.proof ← baseNode.children[0].proof
39:         baseNode.disproof ← baseNode.children[0].disproof
40:         return node
41:       end if
42:     else
43:       bestChild.depth ← node.depth + 1
44:       return SelectMostProving(bestChild)
45:     end if
46:   end if
47: end function

```

- **baseNode** – wierzchołek bazowy rodzica²;
- **pdPresent** – parametr wyjściowy oznaczający, że prawdopodobny jest remis.

²Jeżeli rodzic jest wierzchołkiem bazowym to jest on również wierzchołkiem *baseNode*.

Informuje czy którykolwiek z wierzchołków potomnych jest oznaczony jako *prawdopodobny remis*. Jest inicjalizowany wewnątrz funkcji.

Kryteria wyboru najlepszego wierzchołka potomnego są inne w przypadku rodzica typu OR, a inne w przypadku rodzica typu AND:

- **rodzic typu OR** – najlepszy wierzchołek potomny to taki, który zawiera najniższą wartość liczby dowodzącej. Jeżeli każdy z wierzchołków potomnych jest oznaczony jako *prawdopodobny remis*, lub jest obalony, nie jest zwracany jakikolwiek wierzchołek potomny;
- **rodzic typu AND** – najlepszy wierzchołek potomny to taki który jest oznaczony jako *prawdopodobny remis*, a w przypadku braku takiego wierzchołka ten, który zawiera najniższą wartość liczby obalającej. Jeżeli każdy z wierzchołków potomnych jest oznaczony albo jako *prawdopodobny remis*, albo jest udowodniony, to nie jest zwracany jakikolwiek wierzchołek potomny. Jednak w przypadku kiedy istnieje przynajmniej jeden wierzchołek oznaczony jako *prawdopodobny remis*, wartość parametru wyjściowego *pdPresent* zostaje ustawione na *true*.

Po znalezieniu najbardziej dowodzącego wierzchołka zostaje on rozwinięty poprzez procedurę *DevelopNode(node)* (alg. 24). Podczas rozwijania wierzchołka mogą nastąpić trzy możliwości:

- **wierzchołek bazowy jest rozwiązywany** (linia nr. 7) – wierzchołek bliźniaczy zostaje rozwiązywany do identycznej wartości;
- **wierzchołek został oznaczony jako *prawdopodobny remis*** (linia nr. 10) – wierzchołek zostaje rozwiązywany do wartości reprezentującej rzeczywisty remis¹;
- **wierzchołek jest wierzchołkiem granicznym** (linia nr. 15) – zostają wygenerowane i wyliczone wszystkie jego wierzchołki potomne (natychmiastowe wyliczanie). Następnie każdy wierzchołek potomny, jeżeli nie jest „bliźniakiem”, jest oznaczany jako nierożwinięty. W przeciwnym przypadku atrybut ten został wcześniej ustawiony jako rozwinięty w celu ochrony przed rozwijaniem wierzchołków powtórzonych. Następnie analizowany wierzchołek zostaje od razu wyliczony za pomocą procedury *SetProofAndDisproofNumbers(node)*, a liczby dowodząca i obalająca odpowiadającego mu wierzchołka bazowego zostają zaktualizowane.

Algorytm 23 BTA wybierający najlepszy wierzchołek potomny

```

1: function SELECTBESTCHILD(node, baseNode, OUT pdPresent)
2:   bestChild ← NIL
3:   bestValue ←  $\infty$ 
4:   pdPresent ← false
5:   if node.type = OR then
6:     for all child in baseNode.children do
7:       if child.possibleDraw then
8:         pdPresent ← true
9:       else if child.proof < bestValue then
10:        bestChild ← child
11:        bestValue ← child.proof
12:      end if
13:    end for
14:   else
15:     for all child in baseNode.children do
16:       if child.possibleDraw then
17:         pdPresent ← true
18:         bestChild ← NIL
19:         break
20:       else if child.disproof < bestValue then
21:         bestChild ← child
22:         bestValue ← child.disproof
23:       end if
24:     end for
25:   end if
26:   return bestChild
27: end function

```

Po rozwinięciu wierzchołka zostają uaktualnione wierzchołki przodków, począwszy od rodzica, a skończyszy na korzeniu. W DCG może istnieć wiele równoległych dróg prowadzących do korzenia. Jednakże uaktualniana jest tylko ta ścieżka, poprzez którą został wybrany najbardziej dowodzący wierzchołek. Inne istniejące ścieżki zostaną zaktualizowane podczas innych selekcji. Wszystkie etykiety *prawdopodobny remis* którekolwiek z wierzchołków znajdujących się na aktualizowanej ścieżce są

¹W tym przypadku jest to wartość *false*.

Algorytm 24 BTA rozwijający wierzchołek

```

1: procedure DEVELOPNODE(node)
2:   if node.twinNode then
3:     baseNode ← node.baseNode
4:   else
5:     baseNode ← node
6:   end if
7:   if baseNode.proof = 0 OR baseNode.disproof = 0 then
8:     node.proof ← baseNode.proof
9:     node.disproof ← baseNode.disproof
10:    else if node.possibleDraw then
11:      node.proof ←  $\infty$ 
12:      node.disproof ← 0
13:      baseNode.proof ←  $\infty$ 
14:      baseNode.disproof ← 0
15:    else
16:      GenerateAllChildren(baseNode)
17:      for all child in baseNode.children do
18:        Evaluate(child)
19:        SetProofAndDisproofNumbers(child)
20:        if NOT child.twinNode then
21:          child.expanded ← false
22:        end if
23:      end for
24:      SetProofAndDisproofNumbers(baseNode)
25:      baseNode.expanded ← true
26:      node.proof ← baseNode.proof
27:      node.disproof ← baseNode.disproof
28:    end if
29:  end procedure

```

usuwane, a wartości ich liczb dowodzących i obalających nie są brane pod uwagę w procesie aktualizacji. Wynika to z faktu, że ich uwzględnienie mogłoby błędnie skierować procedurę wyszukującą najbardziej dowodzący wierzchołek do jednego z nich węzłów potomnych, który jest powtórzeniem i nie wnosi żadnych informacji.

Do uaktualniania wierzchołków przodków służy procedura *UpdateAncestors(node)*

Algorytm 25 BTA uaktualniający przodków

```

1: procedure UPDATEANCESTORS(node, root)
2:   while node  $\neq$  NIL do
3:     if node.twinNode then
4:       baseNode  $\leftarrow$  node.baseNode
5:     else
6:       baseNode  $\leftarrow$  node
7:     end if
8:     if node.type = OR then
9:       UpdateOrNode(node, baseNode)
10:    else
11:      UpdateAndNode(node, baseNode)
12:    end if
13:    node  $\leftarrow$  ParentOnCurrentSearchPath(node)
14:   end while
15:   if root.possibleDraw then
16:     root.possibleDraw  $\leftarrow$  false
17:   end if
18: end procedure

```

(alg. 25). Proces uaktualniania przebiega inaczej w przypadku wierzchołka typu OR (linia nr. 9), a inaczej w przypadku wierzchołka typu AND (linia nr. 11).

W przypadku wierzchołka OR uruchamiana jest procedura opisana jako alg. 26. Otrzymuje ona jako parametry uaktualniany wierzchołek i jego wierzchołek bazowy. Algorytm ten jest w zasadzie identyczny jak część procedury *SetProofAndDisproofNumbers(node)* odpowiadająca wierzchołkom OR, za wyjątkiem wspomnianej już sytuacji, kiedy to wierzchołek potomny jest oznaczony jako *prawdopodobny remis*. W takim przypadku wierzchołek potomny jest ignorowany przez tę procedurę, a etykieta *prawdopodobny remis* jest kasowana. Kiedy wierzchołek staje się obalony (przypadek kiedy wszystkie wierzchołki potomne albo są obalone, albo oznaczone jako *prawdopodobny remis*) i istniał wierzchołek potomny z etykietą *prawdopodobny remis*¹ to wartość wierzchołka jest obliczana za pomocą procedury *SetProofAndDisproofNumbers(node)*, w innym przypadku wartość ta jest już wyliczona poprawnie. Jeżeli wierzchołek zostanie rozwiązyany to jego wierzchołek bazowy również staje się rozwiązyany

¹Czas przeszły ponieważ jego etykieta została już usunięta a fakt, że taki wierzchołek istniał przechowywany jest tylko w zmiennej *pdPresent* (alg. 26, linia nr. 7).

Algorytm 26 BTA uaktualniający wierzchołki OR

```

1: procedure UPDATEORNODE(node, baseNode)
2:   min  $\leftarrow \infty$ 
3:   sum  $\leftarrow 0$ 
4:   pdPresent  $\leftarrow \text{false}$ 
5:   for all child in baseNode.children do
6:     if child.possibleDraw then
7:       pdPresent  $\leftarrow \text{true}$ 
8:       proof  $\leftarrow \infty$ 
9:       disproof  $\leftarrow 0$ 
10:      child.possibleDraw = false
11:    else
12:      proof  $\leftarrow \text{child.proof}$ 
13:      disproof  $\leftarrow \text{child.disproof}$ 
14:    end if
15:    if proof < min then
16:      min  $\leftarrow \text{proof}$ 
17:    end if
18:    sum  $\leftarrow \text{sum} + \text{disproof}$ 
19:  end for
20:  if min =  $\infty$  AND pdPresent then
21:    SetProofAndDisproofNumbers(node)
22:  else
23:    node.proof  $\leftarrow \text{min}$ 
24:    node.disproof  $\leftarrow \text{sum}$ 
25:  end if
26:  if node.proof = 0 OR node.disproof = 0 then
27:    baseNode.proof  $\leftarrow \text{node.proof}$ 
28:    baseNode.disproof  $\leftarrow \text{node.disproof}$ 
29:  end if
30: end procedure

```

i wymaga aktualizacji (linie nr. 27 – 28).

W przypadku wierzchołka AND uruchamiana jest procedura opisana jako alg. 27. Ona również otrzymuje jako parametry uaktualniany wierzchołek i jego wierzchołek bazowy. Również ten algorytm jest w zasadzie identyczny jak część proce-

Algorytm 27 BTA uaktualniający wierzchołki AND

```

1: procedure UPDATEANDNODE(node, baseNode)
2:   min  $\leftarrow \infty$ 
3:   sum  $\leftarrow 0$ 
4:   for all child in baseNode.children do
5:     proof  $\leftarrow$  child.proof
6:     disproof  $\leftarrow$  child.disproof
7:     sum  $\leftarrow$  sum + proof
8:     if disproof < min then
9:       min  $\leftarrow$  disproof
10:    end if
11:   end for
12:   node.proof  $\leftarrow$  min
13:   node.disproof  $\leftarrow$  sum
14:   if node.proof = 0 OR node.disproof = 0 then
15:     baseNode.proof  $\leftarrow$  node.proof
16:     baseNode.disproof  $\leftarrow$  node.disproof
17:   end if
18: end procedure

```

dury $SetProofAndDisproofNumbers(node)$ odpowiadająca wierzchołkom AND i jedną różnicą jest to, że w przypadku rozwiązania wierzchołka jego wierzchołek bazowy również zostaje rozwiązany.

3.6. Inne wersje algorytmu „Proof-number search”

Pionki nie mogą wracać na pozycje startowe.

Oryginalny algorytm „Proof-Number Search” doczekał się wielu odmian typu *depth-first search*. Pierwszą z nich jest algorytm PN* [26, 48, 56]. Używa on górnego ograniczenia na liczbę wierzchołków dowodzących – $pn(n)$. Rozpoczyna poszukiwania od korzenia z ograniczeniem: $pn(root)$ równym 2. Jeżeli żadne rozwiązanie nie zostanie znalezione to zaczyna od nowa poszukiwania, zwiększając to ograniczenie o jeden. Zatem w następnym kroku ograniczenie $pn(root)$ będzie wynosiło 3. Proces ten jest kontynuowany aż do znalezienia rozwiązania. Aby zapobiegać odwiedzaniu wcześniejszych wygenerowanych wierzchołków PN* gromadzi dotychczasowe wyniki w tablicy transpozycji. Przed rozwinięciem wierzchołka sprawdzana jest jego liczba dowodząca przechowywana w tablicy. Jeżeli wpis istnieje to wartość liczby dowodzą-

cej jest od razu inicjowana magazynowaną wartością, w innym przypadku ustawiana jest na 1. Jedną z niedoskonałości tego algorytmu jest brak współpracy z liczbami obalającymi, co może powodować pewne problemy podczas rozwijania węzłów typu AND.

Tej niedoskonałości pozbawiony jest algorytm PDS [26, 48, 64, 65], który jest rozwinięciem PN*. Wykorzystuje on dwa ograniczenia – $pn(n)$ oraz $dn(n)$ i rozwija wierzchołki dopóki nie przekroczą obydwiu wartości. PDS jest zatem użyteczny zarówno do dowodzenia drzewa, jak i do jego obalania.

Kolejnym rozszerzeniem jest algorytm $Df\text{-}pn^1$ [26–28, 35], najwyraźniej widoczną w nim różnicą jest zasięg ograniczeń $pn(n)$ i $dn(n)$ który został zmieniony z globalnego (jak to miało miejsce w PN* i PDS) do lokalnego. Dzięki temu jest on w stanie uzyskać te same wyniki rozwijając mniejszą liczbę węzłów od swoich poprzedników. Wszystkie z tych algorytmów cechują mniejsze wymaganiem pamięciowe, jednak zazwyczaj odbywa się to kosztem wykorzystanego czasu.

Ciekawostką jest również algorytm „Monte-Carlo Proof-Number Search” [47], który jest fuzją algorytmów „Monte-Carlo” i „Proof-Number Search”. Podczas wyliczania węzła, jeżeli nie zaostanie on rozwijany, przypisywana mu jest wartość z przedziału $(0, 1]$. Wartość tę należy interpretować jako przewidywane prawdopodobieństwo znalezienia rozwiązania. Jest ona oszacowywana przy wykorzystaniu algorytmu „Monte Carlo” i musi być większa od 0 aby uniknąć fałszywych rozwiązań w których liczba dowodząca albo obalająca przyjmie wartość 0 błędnie oznaczającą rozwiązanie albo obalenie wierzchołka.

Niestety ograniczenia ilościowe niniejszej pracy nie pozwalają na dalszy szczegółowy opis wspomnianych algorytmów, w przypadku zainteresowania nimi proponowane jest czytelnikowi zapoznanie się z powyżej podanymi pozycjami z bibliografii.

¹Nagai's depth-first proof-number algorithm.

Rozdział 4

Gra Skoczki

4.1. Dlaczego Skoczki?

Podczas wyboru gry będącej celem przeprowadzonych badań, w praktycznej części tej pracy, jako źródło posłużył „Przewodnik Gier” – Lecha Pijanowskiego [40].

Brane pod uwagę (zgodnie z przyjętą konwencją) były jedynie gry należące do określonego podzbioru gier dwuosobowych, deterministycznych, z pełną informacją, szczegółowo opisanego pod koniec sekcji 2.1 roz. 2. Pominięto zatem gry z elementami losowymi (przykładowo: *Backgamon*, *Chińczyk*, *Bitwa morska*¹, różne warianty *Domina* oraz cały rozdział opisujący gry w karty). Również dwa ostatnie rozdziały tej książki: „Gry dla samotnych” i ”*Pasjanse*” nie zakwalifikowały się już na początku selekcji, ponieważ dotyczą tylko gier jednoosobowych.

W pierwszym kroku zostały odrzucone gry już wcześniej rozwiązane, takie jak: *Czwórki*, *Go-Moku*, *Młynek*, *Pentomino*, *Dakon*, *Kalah*, *Renju*, *Avari*, omówione w roz. 2, sekcja 2.3, aby przysłowiowo „nie wyważać otwartych drzwi”.

Z oczywistego powodu zostały wyeliminowane takie gry jak: *Othello*, *Warcaby polskie*, *Szachy*, *Hex* czy *Go*, opisane w sekcji 2.4, jak również ich liczne warianty. Nie ulega bowiem wątpliwości, że są to gry na tyle trudne, iż rozwiązanie ich dzisiaj dostępnymi środkami jest niemożliwe.

Z pozostałych gier autor wybrał te, które mają pewne pozytywne właściwości pozwalające na skalowanie gry i efektywną implementację jej generatora posunięć².

Gry skalowalne to takie, w których możliwa jest zmiana rozmiarów planszy, po której nadal można grać w tę grę według nie zmienionych zasad. Najlepsze okazują się plansze prostokątne z jak najmniejszą liczbą cech różniących pola gry (pola startowe, pola przemiany, inne specjalne pola).

Generatory posunięć są tym szybsze, im zasady gry odpowiadające za posunię-

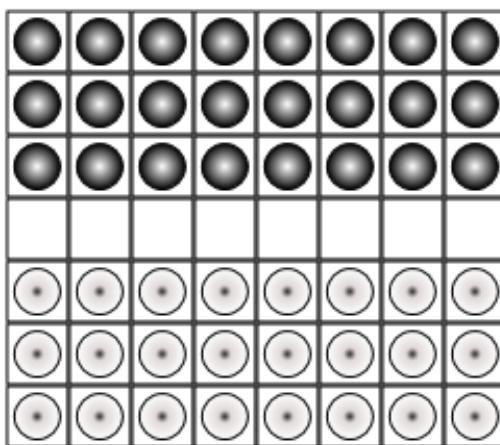
¹Znana pod nazwą *Gra w statki*.

²Dobrym kontrprzykładem takiej gry jest *Dżungla* [40], w której istnieje osiem rodzajów bierek (szczur, kot, wilk, pies, pantera, tygrys, lew i słoń) oraz pięć rodzajów pól (zwykłe, startowe, jama, jezioro, pułapka). Możliwości posunięć w tej grze uzależnione są od rodzaju bierki oraz pola na jakim się ona znajduje, a w przypadku bicia dodatkowo od rodzaju bierki przeciwnika i pola na jakim ta bierka się znajduje.

cia są prostsze i bardziej niezależne od dodatkowych czynników, takich jak: rodzaj bierki, rodzaj bierki przeciwnika, rodzaj pola itp.

Zmiana rozmiaru planszy, ma wpływ na wielkość problemu obliczeniowego jaki reprezentuje rozwiązywanie danej gry. Można, zaczynając od najmniejszych możliwych rozmiarów planszy, wykonać pewne badania sprawdzając przy okazji poprawność napisanych algorytmów (generatora ruchów, funkcji wykrywającej koniec rozgrywki, algorytmów rozwiązujących grę). Będąc bogatszym o te doświadczenia można (zwiększając rozmiary planszy) kontynuować badania, zmagając się z coraz to większymi wyzwaniami stawianymi nam przez tę samą grę. Biorąc od uwagi powyższe fakty, najbardziej odpowiednie wydawały się cztery pozycje:

- *Wilc i owce*¹ – gra o niesymetrycznych zasadach, grana na tradycyjnej warcabnicy (8×8). Cechuje ją mała liczba bierek, co oznacza niewielką złożoność przestrzeni stanów gry. Również głębokość drzewa gry jest skończona i niewielka. Dobra na pierwsze starcie jako rozgrzewka przed wzmaganiem się z trudniejszymi problemami. Najprawdopodobniej została już wcześniej rozwiązana.
- *Latrunculi*² – starorzemska gra, rozgrywana na prostokątnej planszy o rozmiarach (8×7) przedstawionej na rys. 4.1. Przejrzyste zasady łatwo dające się



RYSUNEK 4.1. Plansza do gry *Latrunculi* zawierająca pozycję startową

skalować. Jest to gra zbieżna (dopuszcza bicie pionków przez pojmanie³).

- *Skoczki*¹ – rozgrywana na planszy o większych rozmiarach (6×11), łatwo

¹Zasady gry *Wilc i owce* przedstawiono w dodatku B.27.

²Zasady gry *Latrunculi* przedstawiono w dodatku B.14.

³Bicie przez pojmanie odbywa się w wyniku ustawienia dwóch własnych pionów po obydwu stronach nieprzerwanego rzędu bądź kolumny pionów przeciwnika.

¹Zasady gry *Skoczki* przedstawiono w sekcji 4.2.

skalowalna, posiada nieskomplikowane zasady gry. W grze występuje zawsze stała liczba pionków (nie ma bić).

- *Halma*² – podobna gra do Skoczków, rozgrywana na planszy o większych rozmiarach (16×16), i z większą liczbą pionków. Każdy pionek ma większą liczbę możliwości wykonania posunięcia, co zwiększa współczynnik rozgałęzienia drzewa tej gry. Wydaje się ona naturalnym kandydatem do wzmagania po rozwiązaniu gry *Skoczki*. Gra stanowi bardzo duże wyzwanie, a w połączeniu z jej popularnością, rozwiązanie tej gry mogłoby zapewnić jego autorowi spory sukces.

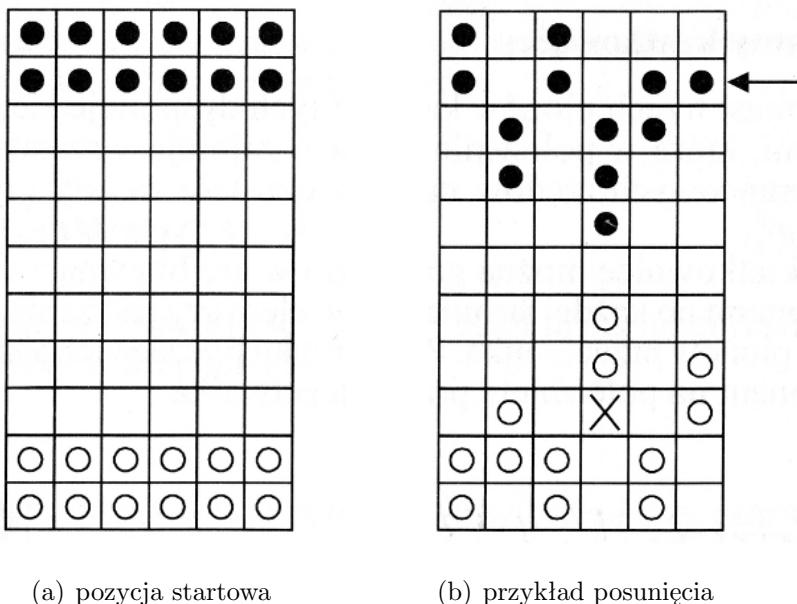
Spośród nich została wytypowana jednak gra *Skoczki*, ponieważ jest ona grą łatwo skalowalną, posiadającą nieskomplikowane zasady. Jest też podobna do popularnej *Halmy* bądź jej trójosobowego wariantu *Trylmy*, a jest mniej złożona od nich.

4.2. Oryginalne zasady gry

Zasady gry *Skoczki*, opisane przez Lecha Pijanowskiego są następujące:

- Rozgrywka toczy się na prostokątnej planszy o wymiarach 6 kolumn \times 11 rzędów.
- Każdy z graczy dysponuje dwunastoma pionkami – jedne w białym, a drugie w czarnym kolorze.
- Początkowo pionki ustawione są w dwóch skrajnych rzędach po przeciwnielego stronach planszy rys. 4.2(a).
- Grę rozpoczyna grający pionkami białymi.
- Gracze wykonują posunięcia na zmianę.
- Ruch pionka odbywa się tylko w pionie bądź poziomie, nigdy po skosie.
- Ruch pionka polega na przeskoczeniu dowolnego pionka, bądź nieprzerwanego bloku pionków. Mogą to być zarówno pionki własne jak i przeciwnika, lub dowolna ich kombinacja.
- Ruch kończy się na pierwszym wolnym polu za przeskakiwanymi pionkami.

²Zasady gry *Halma* przedstawiono w dodatku B.10.



(a) pozycja startowa

(b) przykład posunięcia

RYSUNEK 4.2. Plansza gry *Skoczki* (wg L.Pijanowskiego) [40]

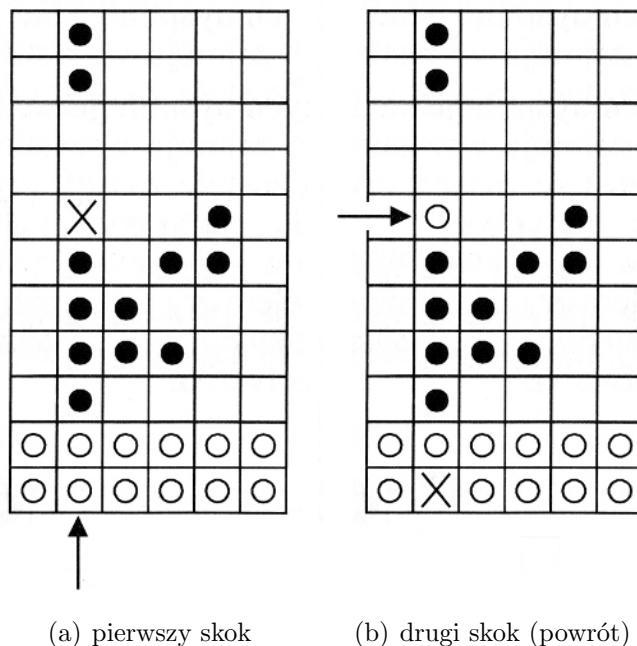
- Ruch jest obowiązkowy o ile gracz ma możliwość go wykonać, w przeciwnym razie traci kolejkę.
- Jeśli po wykonaniu ruchu pionkiem istnieje możliwość kolejnego ruchu w dowolnym kierunku to można (ale nie trzeba) taki ruch wykonać.
- Wygrywa ten z graczy, któremu jako pierwszemu uda się zająć swoimi pionkami wszystkie startowe pozycje przeciwnika.

Rys. 4.2(b) przedstawia przykładowe posunięcie. Grający czarnymi może w jednym posunięciu pionkiem (wskażanym przez strzałkę) przeskoczyć (w lewo) dwa własne piony, następnie (zmieniając kierunek ruchu w dół) przeskoczyć kolejno trzy własne i dwa piony przeciwnika, osiągając tym pozycję zaznaczoną krzyżykiem.

4.3. Poprawione zasady gry

Zasady podane w poprzedniej sekcji pozwalają na zastosowanie przez każdego z graczy strategii która łatwo pozwala na uniknięcie przegranej. Zasady pozwalają na wykonanie jednego bądź większej liczby przeskoków w jednym posunięciu i nie zabraniają zajmowania przez pionek pozycji z której nastąpiło posunięcie. Zasady tej gry są zatem tak sformułowane, że po wykonaniu dowolnego skoku zawsze można wykonać skok odwrotny zajmując pozycję startową. Wystarczy zatem aby gracz wykonywał,

w każdym swoim posunięciu ruch tylko pionkami z zewnętrznego rzędu planszy. Ruch taki będzie polegał na przeskoczeniu własnego piona i ewentualnych pionów przeciwnika, a następnie powrocie na początkową pozycję startową. Przykład takiego posunięcia przedstawiam na rys. 4.3. Przynajmniej jedno takie posunięcie zawsze



RYSUNEK 4.3. Przykładowe posunięcie wg trywialnej strategii remisowej

jest możliwe, ponieważ przeciwnik nie może zablokować wszystkich kolumn planszy. Musiałby zastawić wszystkie wolne pola, ale wtedy również by nie wygrał z powodu niemożności wykonania ruchu przez któregokolwiek z graczy.

Jeśli założymy że gracz nie może kończyć posunięcia na polu startowym to zasady stają się ciekawsze, jednak dalej istnieje ryzyko wykorzystywania przez któregokolwiek z graczy prostej strategii remisowej. Polega ona na pozostawieniu (i nie opuszczaniu) jednego bądź większej liczby pionków na własnych pozycjach startowych. Zajęte pozycje nie zostaną zwolnione przez cały czas trwania rozgrywki. Jeśli którykolwiek z graczy podejmie taką strategię to przeciwnik nie będzie miał możliwości usunięcia tych pionków z pozycji startowej gracza¹, co uniemożliwia jego wygraną. Wynika to z faktu, że w grze nie można zbijać pionków, oraz że określony ruch na przeciwniku można wymusić jedynie w przypadku pozbawienia wszystkich pozostałych pionków możliwości wykonania posunięcia, a jedyne mobil-

¹Pamiętamy, że pola startowe jednego z graczy są zarazem polami końcowymi jego przeciwnika.

nemu pionkowi pozostawić możliwość wykonania jedynie interesujących nas skoków (opuszczających pozycję startową). O ile teoretycznie zablokowanie wszystkich pionków gracza jest możliwe, to przy optymalnie prowadzonej przez niego taktyce wydaje się to niemożliwe. A zatem, jeśli obaj gracze posiadają strategię remisującą to żaden z nich nie będzie starał się wygrać, bowiem zakładając, że przeciwnik gra optymalnie wiemy, iż wybrał najlepszą dla siebie strategię, czyli remisującą. W pierwszym kroku doświadczeń należało by wykazać (w formie dowodu albo empirycznie) że taka strategia jest słuszna. W przypadku jej dowodzenia wiedzielibyśmy że rozwiązywanie gry z takimi przepisami nie ma sensu, ponieważ z góry znamy zarówno wynik jak i optymalną strategię gry (słabe rozwiązywanie).

Aby gra stała się nietrywialna i ciekawa z punktu widzenia algorytmicznego należałyby dodać reguły eliminujące wcześniej opisany mankament. Jedną z możliwych propozycji takiego wzbogacenia zasad gry jest rozszerzenie ich o dwie następujące zasady:

- Jeśli którykolwiek z pionków przeciwnika ma możliwość opuścić pozycję startową to taki ruch jest obowiązkowy.
- Pionki nie mogą wracać na pozycje startowe.

Tę drugą zasadę można interpretować w dwojakim sposobie:

- Pionki, nawet podczas wykonywania serii przeskoków, nie mogą wykorzystywać własnych pozycji startowych.
- Pionki, nie mogą kończyć posunięcia na pozycji startowej, mogą natomiast zajmować je podczas wykonywania serii przeskoków.

Druga interpretacja tej zasady jest mniej restrykcyjna i czasami nieznacznie zwiększa liczbę możliwych posunięć graczy.

4.4. Inne warianty gry

Inny wariant tej gry, na jaki można natrafić, jest rozgrywany na warcabnicy o rozmiarach (8×8) . Planszę do tej wersji gry wraz z ustalonymi pionkami w pozycji początkowej przedstawiono na rys. 4.4.

Generalnie zasady tej wersji gry nieznacznie różnią się od wersji Pijanowskiego. Jedyną różnicą (poza innym rozmiarem planszy) jest dodatkowa możliwość związana z przemieszczaniem pionków. Mianowicie mogą one zamiast skoku wykonywać „normalne” posunięcie o jedno pole w pionie bądź poziomie. Takie posunięcie automatycznie kończy turę gracza¹. W tym przypadku udowodnienie posiadania przez każdego

¹Inaczej niż w przypadku skoku, po którym można kontynuować przeskakiwanie.



RYSUNEK 4.4. Plansza do gry *Skoczki*

z gracz strategii remisowej jest już bardzo łatwe, ponieważ zablokowanie pionków przeciwnika jest niemożliwe. Wynika to z faktu, że aby zablokować wszystkie (bez jednego) pionki gracza, przeciwnik musiałby wypełnić swoimi pionkami wszystkie rzędy i kolumny w których znajduje się jakikolwiek pionek gracza, a na to nigdy nie wystarczy mu pionków.

Autor nie założył sztywnego rozmiaru planszy oraz „jedynych słuszych” możliwości posunięć pionków. Jak nadmieniono na początku rozdziału, można wykorzystać skalowalność tej gry, rozpoczynając badania od najmniejszych możliwych rozmiarów planszy. Na tym etapie są na tyle proste, że daje się je przeprowadzić na kartce papieru albo w pamięci. Ich zaletą test to, że pozwalają poznać naturę gry oraz prześledzić poprawność algorytmów. Natomiast dla jakich maksymalnych rozmiarów planszy uda się znaleźć odpowiedź określającą wartości gry, było wielką niewiadomą w momencie rozpoczęcia badań.

Dodatkowym przedmiotem badań było sprawdzenie, jak mobilność pionków ma wpływ na wynik i złożoność gry. W tym celu wykorzystano mieszane warianty, w których:

- dopuszczalne są albo tylko pojedyncze skoki (spodziewany mniejszy współczynnik rozwidlenia), albo można wykonywać dowolną ich liczbę (oczywiście jeżeli istnieje taka możliwość);
- pionki, podczas wykonywania serii przeskoków nie mogą wykorzystywać własnych pozycji startowych (spodziewany nieco mniejszy współczynnik rozwidlenia), bądź możliwe są wielokrotne skoki, które nie mogą kończyć posunięcia na pozycji startowej;

- obok skoków możliwe są posunięcia (spodziewany większy współczynnik rozwinięcia).

Wyniki przeprowadzonych eksperymentów przedstawiono w roz. 6.

Rozdział 5

Program Weles

Program Weles pozwala na przeprowadzenie rozgrywki w grę *Skoczki* pomiędzy dwoma graczami. Graczem może być człowiek lub komputer. Pozwala na wykorzystanie różnych wariantów tej gry, które mogą różnić się zasadami dotyczącymi ruchów pionków oraz rozmiarami planszy.

5.1. Środowisko programistyczne

Program został napisany przy wykorzystaniu IDE¹ Microsoft Visual Studio 2008 w języku C#. Do jego uruchomienia wymagane jest środowisko .NET Framework w wersji 3.5 (zamieszczone na dołączonej płycie DVD). Graficzny interfejs użytkownika bazuje na bibliotece WPF².

5.2. Architektura programu

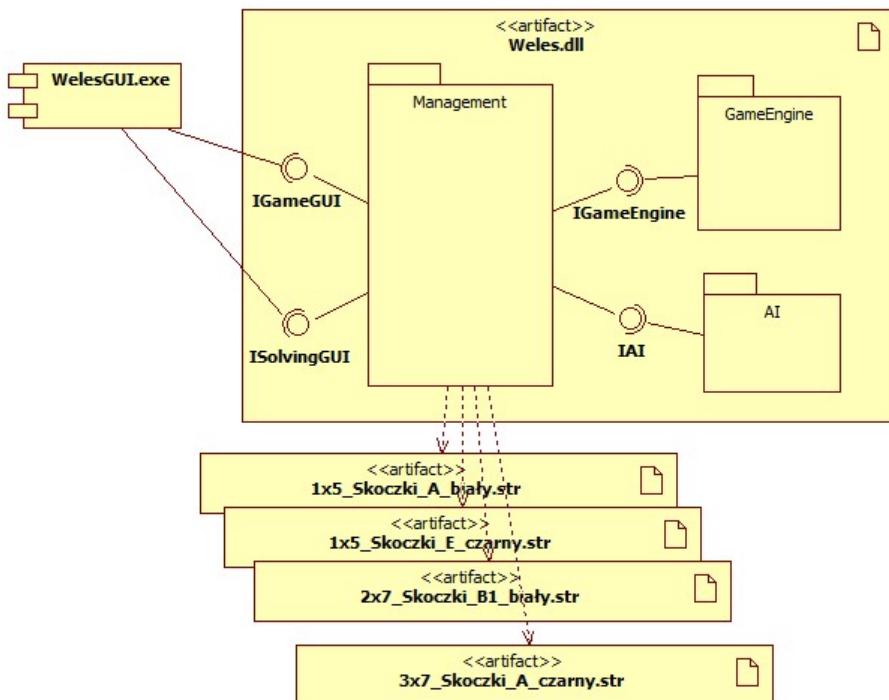
Program składa się z pliku wykonywalnego, biblioteki oraz kilku dodatkowych plików zawierających optymalną strategię graczy dla kilku wariantów gry. Diagram komponentów został uwidoczniony na rys. 5.1. Głównym komponentem jest biblioteka Weles.dll i składa się ona z trzech pakietów. Każdy pakiet reprezentuje osobny moduł gry i jest zamieszczony w osobnej przestrzeni nazw:

- **moduł zarządzania grą** (*Weles.Management*), udostępnia cztery interfejsy:
 - *IGameGUI* – logika interfejsu graficznego służącego do grania w grę;
 - *ISolvingGUI* – logika interfejsu graficznego służącego do rozwiązywania gry;
 - *IGameEngine* – zbiór metod, które musi obsługiwać silnik gry;
 - *IAI* – zbiór metod, które musi obsługiwać algorytm rozwiązymyjący grę.

Moduł może tworzyć a następnie zapisywać w plikach optymalną strategię każdego z graczy dla różnych wariantów gry.

¹Akronim od (ang. *Integrated Development Environment*) – zintegrowane środowisko programistyczne.

²Akronim od (ang. *Windows Presentation Foundation*) – nazwa kodowa *Avalon*.



RYSUNEK 5.1. Diagram komponentów programu Weles

- **moduł silnika gry** (*Weles.GameEngine*), realizuje interfejs *IGameEngine*. Jest jedynym modułem zawierającym informacje o regułach danej gry, dzięki temu jego wymiana pozwala na łatwą zmianę jej zasad i rozwiązywanie różnych wariantów tej gry.
- **moduł sztucznej inteligencji** (*Weles.AI*), realizuje interfejs *IAI*. Pozwala na rozwiązywanie gry za pomocą różnych metod algorytmicznych. W opisywanym programie wykorzystano następujące algorytmy:
 - *analizy wstępnej* – omówiony w sekcji 3.2
 - α - β – omówiony w sekcji 3.3.3
 - *pn-search*¹ współpracujący z drzewem And/Or – omówiony w sekcji 3.5.1
 - *pn²-search* przy współpracy z drzewem And/Or – omówiony w sekcji 3.5.3
 - *pn-search* współpracujący z DCG – omówiony w sekcji 3.5.6

Moduł sztucznej inteligencji może przechowywać swoje obliczenia w słown-

¹Pełna nazwa: *Proof-Number Search*.

ikach¹, co zabezpiecza przed powtórnym wyliczaniem wcześniej rozwiązanych stanów gry i znacznie przyśpiesza pracę programu.

Wspomniany osobny plik wykonywalny to interfejs użytkownika. Zgodnie z założeniem wzorca MVP² nie posiada żadnej logiki. W kolejnych sekcjach tego rozdziału omówiona zostanie szczegółowo rola poszczególnych modułów wraz z opisem ważniejszych klas.

5.3. Interfejs użytkownika

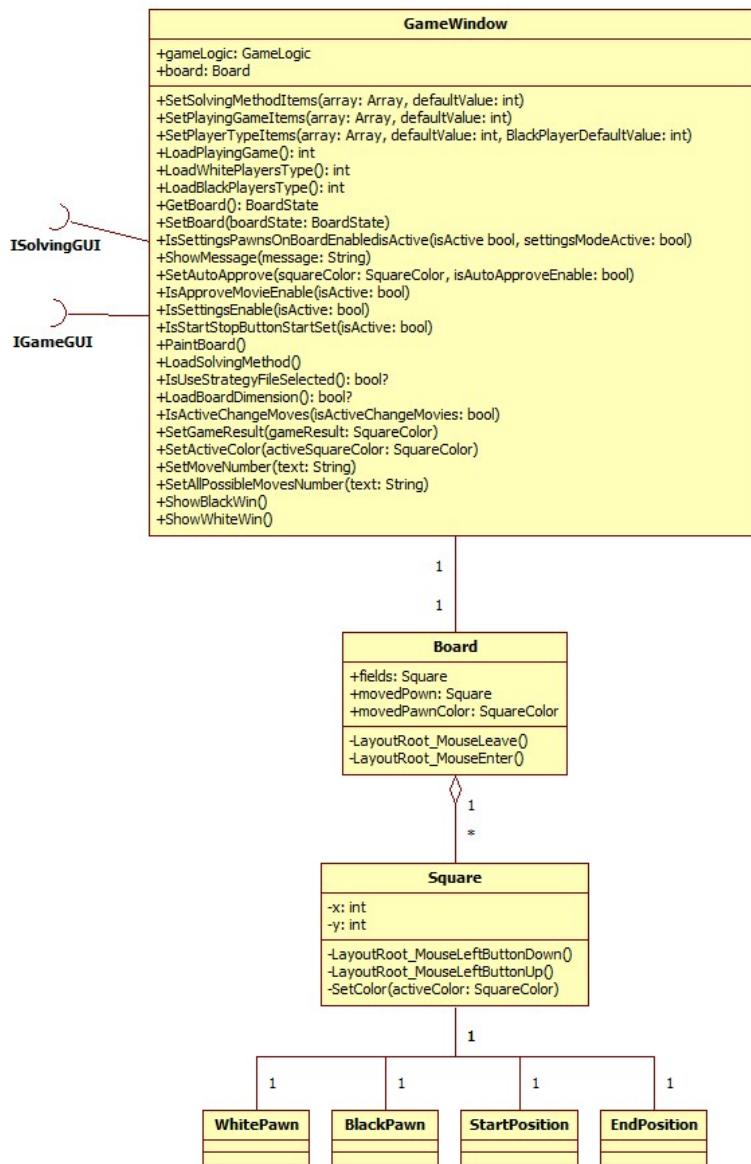
Jak wcześniej nadmieniono, interfejs użytkownika nie posiada żadnej logiki. Została ona wyniesiona do klasy *GameLogic* zamieszczonej w pakiecie *Management*. Pozwala to na łatwą zmianę interfejsu na inny oraz znacznie łatwiejsze testowanie poprawności wydzielonej logiki. W jego skład wchodzą trzy najważniejsze klasy:

- **Game Window** – podstawowa klasa GUI reprezentująca całe okno programu. Realizuje wszystkie zdarzenia zdefiniowane w interfejsie *IGameGUI* oraz *ISolvingGUI*.
- **Board** – klasa reprezentująca wyświetlana planszę do gry. W jej skład wchodzą jedynie pola planszy reprezentowane przez klasę *Square*. Liczba tych pól jest zmienna i zależy od wprowadzonych przez użytkownika rozmiarów planszy;
- **Square** – reprezentuje pole wchodzące w skład klasy *Board*. Każde pole posiada swoje współrzędne oraz stan. Na stan pola składa się stan widoczności czterech klas wchodzących w jego skład:
 - **WhitePawn** – pole zajęte przez biały pionek;
 - **BlackPawn** – pole zajęte przez czarny pionek;
 - **StartPosition** – pole, z którego aktualnie przesuwany pionek rozpoczął posunięcie;
 - **EndPosition** – pole, na którym, aktualnie przesuwany pionek zakończył posunięcie.

Ponadto klasa *Square* implementuje dwa zdarzenia umożliwiające człowiekowi wykonywanie posunięć:

¹Słowniki są standardowymi kolekcjami środowiska .NET pozwalającymi na szybkie wyszukiwanie danych, ponieważ są zaimplementowane jako tablice haszujące *Dictionary(TKey, TValue)* albo jako binarne drzewa poszukiwań *SortedDictionary(TKey, TValue)*.

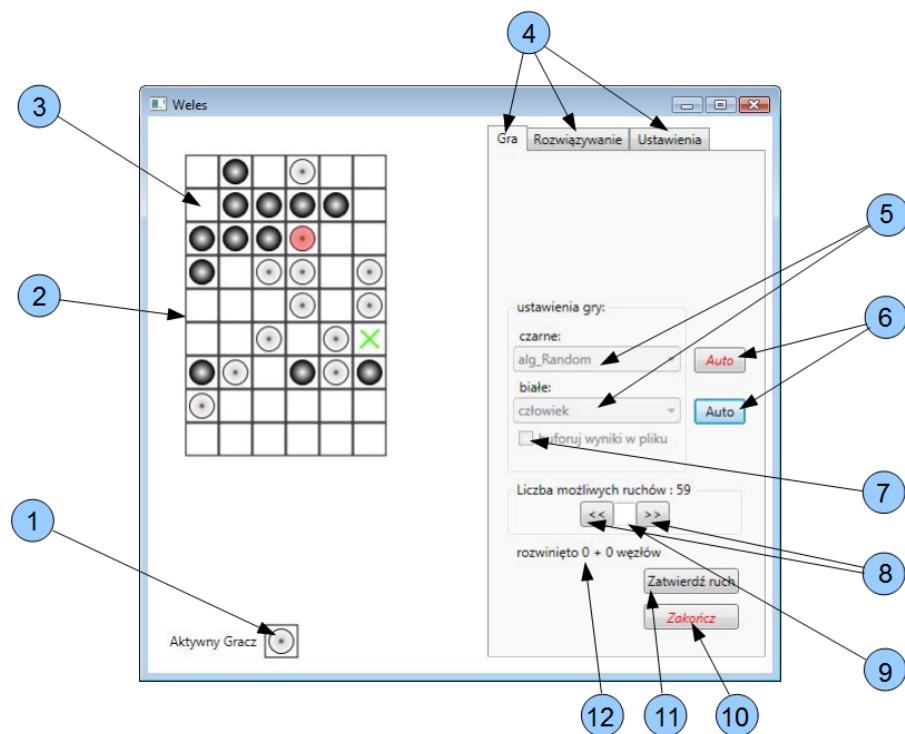
²Akronim od Model-View-Presenter.



RYSUNEK 5.2. Diagram klas interfejsu użytkownika

- *LayoutRoot_MouseLeftButtonDown()* – naciśnięcie prawego przycisku myszy na danym polu;
- *LayoutRoot_MouseLeftButtonUp()* – zwolnienie prawego przycisku myszy na danym polu.

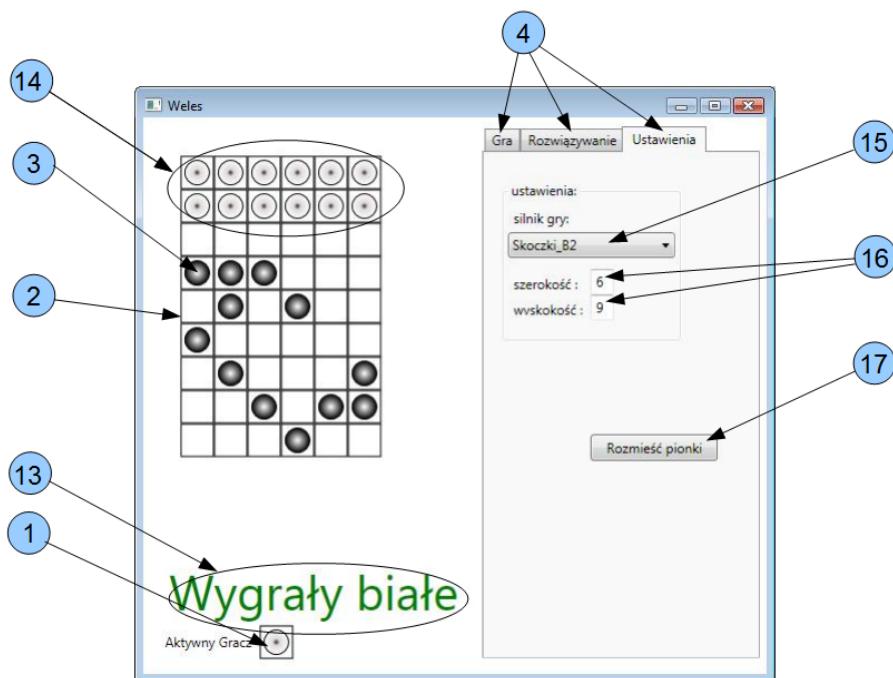
Na rys. 5.5 i 5.4 przedstawiam widok interfejsu jaki może wystąpić podczas trwania oraz na końcu rozgrywki pomiędzy człowiekiem i graczem komputerowym.



RYSUNEK 5.3. Opis interfejsu użytkownika – trwająca rozgrywka

Poniżej szczegółowo opisano wyszczególnione elementy interfejsu:

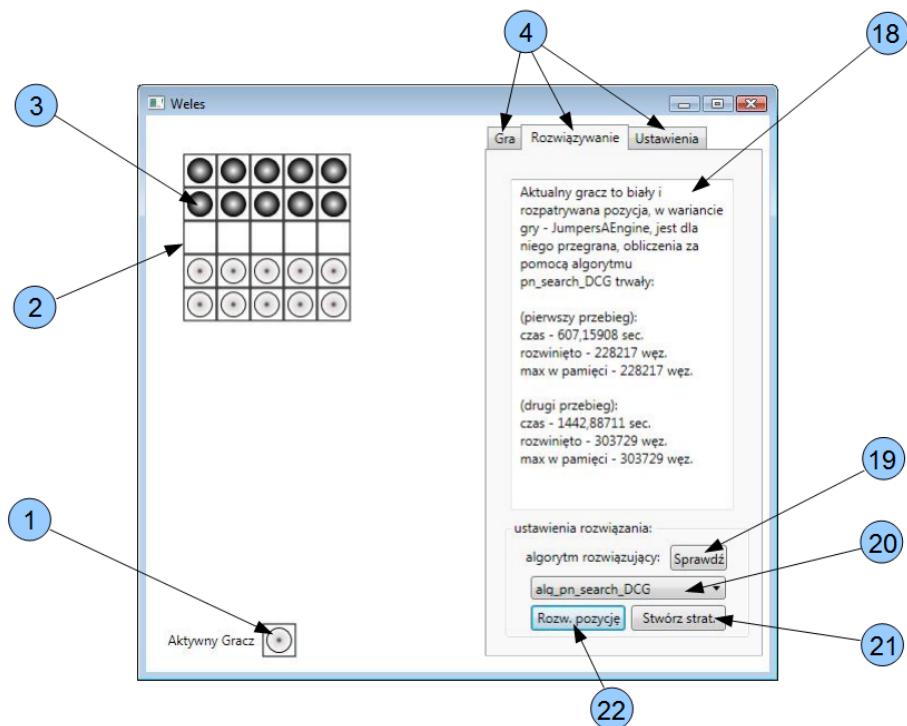
1. Kontrolka informująca, do którego koloru gracza należy aktualne posunięcie.
2. Plansza gry reprezentowana poprzez klasę *Board*.
3. Pole planszy reprezentowane przez klasę *Square*.
4. Zakładki grupujące kontrolki w trzy grupy tematyczne związane z graniem, rozwiązywaniem oraz ustawieniami.
5. Kontrolki wyboru typu gracza białego i czarnego. Graczem może być człowiek lub algorytm.
6. Przyciski automatycznej akceptacji posunięcia.
7. Pole wyboru pozwalające na wykorzystanie pliku przechowującego dotychczasowe obliczenia, pozwala na uniknięcie powtórnych obliczeń.
8. Przyciski pozwalające na wybór jednego z kilku możliwych posunięć aktualnego gracza.



RYSUNEK 5.4. Opis interfejsu użytkownika – wygrana białych

9. Numer kolejnego możliwego posunięcia gracza.
10. Przycisk rozpoczętajacy i przerywajacy aktualna rozgrywke.
11. Przycisk zatwierdzajacy aktualna pozycje na planszy jako posunięcie aktywnego gracza.
12. Liczba węzłów rozwiniętych podczas przeszukiwania przez wybrany algorytm rozwiązujący.
13. Etykieta prezentująca wynik rozgrywki.
14. Końcowa, wygrywająca pozycja białych.
15. Kontrolka wyboru silnika gry.
16. Pola pozwalające na ustalenie rozmiaru planszy.
17. Przycisk pozwalający na zmianę ustawienia pionków na planszy.
18. Konsola wyświetlająca wyniki obliczeń.

19. Przycisk sprawdzania poprawności algorytmu.
20. Kontrolka wyboru silnika rozwiązywającego.
21. Przycisk tworzenia pliku strategii.
22. Przycisk rozwiązywania aktualnej pozycji.



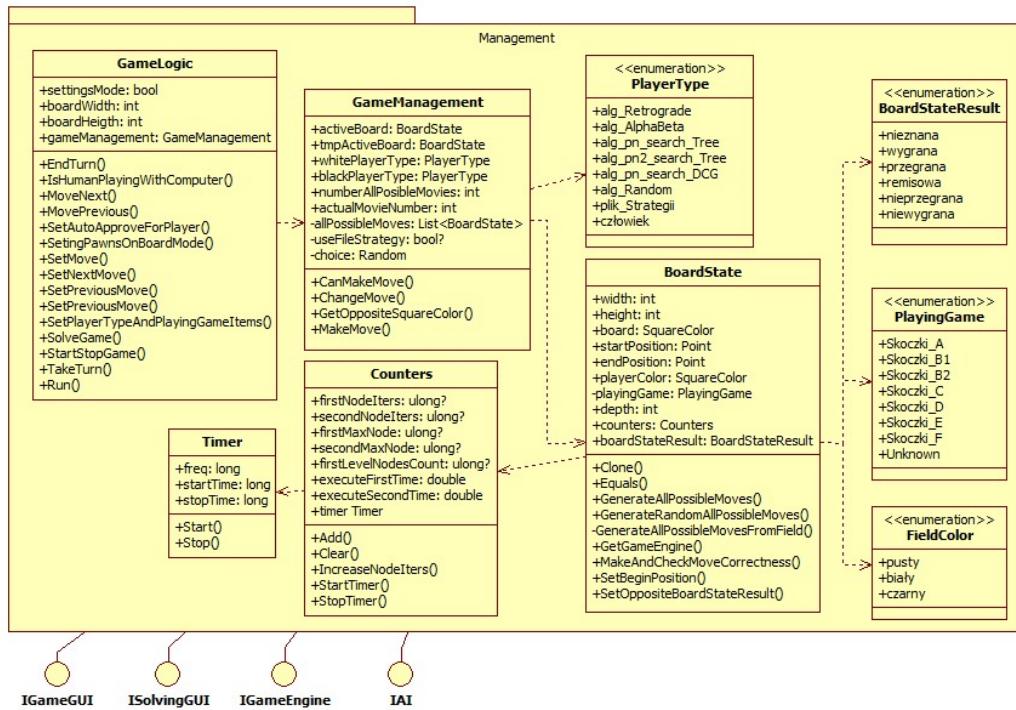
RYSUNEK 5.5. Opis interfejsu użytkownika – rozwiązywanie pozycji startowej

5.4. Moduł zarządzania

Szczegółowy podział modułu zarządzania na klasy został przedstawiony na rys. 5.6.

Trzy najważniejsze z nich to:

- ***GameLogic*** – klasa zawierająca logikę interfejsu użytkownika w postaci scenariusza rozgrywki lub rozwiązyania gry. Komunikacja pomiędzy nią a interfejsem użytkownika odbywa się:
 - od ***GameLogic*** do ***GUI*** – poprzez wywoływanie zdarzeń zdefiniowanych w *GameLogic*, a obsługiwanych w *GUI* w metodach zdefiniowanych poprzez interfejsy *IGameGUI* oraz *ISolvingGUI*.

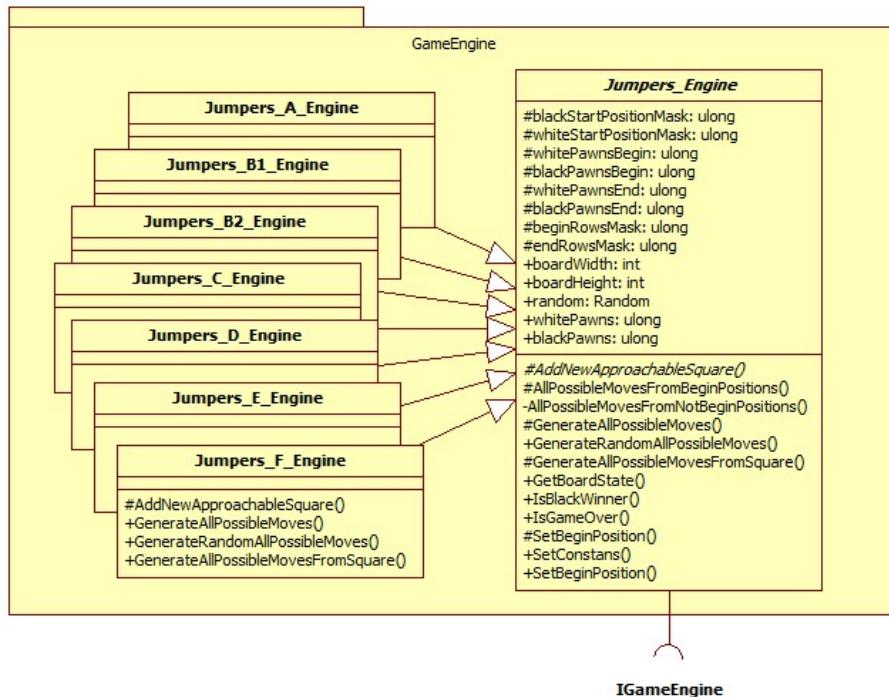


RYSUNEK 5.6. Diagram klas modułu zarządzania

- **od GUI do GameLogic** – poprzez wywoływanie w interfejsie użytkownika publicznych metod klasy *GameLogic*.
- **GameManagement** – klasa wykonująca posunięcie gracza. Podczas posunięć gracza komputerowego wykorzystywany jest odpowiadający mu algorytm. Realizacja tego algorytmu dokonywana jest poprzez metodę zdefiniowaną w interfejsie *IAI*.
- **BoardState** – klasa przechowująca stan gry. Pozwala na generowanie listy możliwych posunięć oraz sprawdzenie, czy przechowywana przez nią pozycja nie jest pozycją końcową. Jej działanie jest ściśle powiązane z modelem silnika gry, a jej zachowanie jest uzależnione od typu wybranego silnika (generatorka posunięć). Współpraca ta jest realizowana poprzez wykorzystanie metod zdefiniowanych w interfejsie *IGameEngine*. Klasa *BoardState* sama w sobie nie jest powiązana z zasadami którejkolwiek z gier, co czyni ją uniwersalną i uniezależnia klasy *GameLogic* i *GameManagement* od sposobu implementacji generatora posunięć.

5.5. Generator ruchów – moduł silnika gry

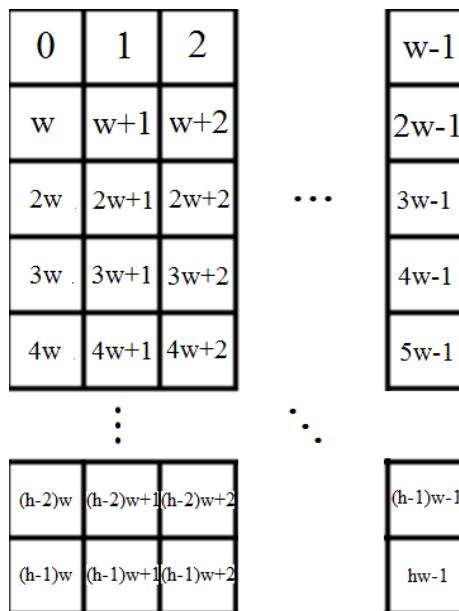
Szczegółowy podział modułu silnika gry na klasy został przedstawiony na rys. 5.7.



RYSUNEK 5.7. Diagram klas modułu silnika gry

Klasa abstrakcyjna *JumperEngine* zawiera dwa pola statyczne typu całkowitego służące do przechowywania rozmiarów planszy oraz dwa pola przechowujące stan gry. Na każdy stan gry składają się dwie informacje. Pierwsza zawiera informację o rozmieszczeniu pionów białych na planszy, a druga o rozmieszczeniu pionów czarnych. Informacje te zapisane są we wspomnianych dwóch polach nazwanych odpowiednio: *whitePawns* i *blackPawns*. Sposób interpretacji tych informacji jest bardzo prosty – bit ustawiony (jedynka logiczna) oznacza, że na danym polu znajduje się pionek, a bit wyzerowany (zero) oznacza, że dane pole takiego pionka nie zawiera. Na rys. 5.8 przedstawiono jak kolejne bity typu całkowitego powiązane są z polami planszy. Zmienna *w* oznacza aktualną szerokość, a *h* aktualną wysokość planszy. Jak można łatwo zauważyć, taki sposób przechowywania informacji o pionkach na planszy może być wykorzystany dla innych podobnych gier planszowych.

W klasie *JumperEngine* zamieszczone są pola i metody, które powiązane są jedynie z zasadami gry *Skoczki* i nie mają uniwersalnego charakteru. Znajdują się tutaj implementacje tylko tych metod, które są wspólne dla wszystkich wariantów tej gry.

RYSUNEK 5.8. Powiązanie pól planszy z bitami typu *ulong*

Jest ona również klasą, po której dziedziczą klasy reprezentujące konkretne wersje silników gry *Jumpers*. Klasy te zawierają jedynie funkcje uwzględniające różnice pomiędzy kolejnymi wersjami gry. W tab. 5.1 przedstawiono zaimplementowane silniki gry oraz ich właściwości. Poniżej opisano każdą z tych właściwości:

TABLICA 5.1. Sześć wariantów gry *Skoczki*

symbol	możliwość:			
	pojedynczego posunięcia	wieloskoku	powrotu na pozycje startowe	przeskoku przez pozycje startowe
A	nie	nie	nie	bez znaczenia
B-1	nie	tak	nie	nie
B-2	nie	tak	nie	tak
C	nie	nie	tak	bez znaczenia
D	nie	tak	tak	bez znaczenia
E	tak	nie	tak	bez znaczenia
F	tak	tak	tak	bez znaczenia

- **możliwość pojedynczego posunięcia** – możliwość wykonywania posunięcia pionkiem na sąsiednie wolne pole (w poziomie bądź w pionie);

- **możliwość wieloskoku** – możliwość wykonywania kilku kolejnych przeskoków jako jedno posunięcie gracza (jak w oryginalnych zasadach gry);
- **możliwość powrotu na pozycje startowe** – możliwość zakończenia posunięcia na własnym polu startowym (jak w oryginalnych zasadach gry);
- **możliwość przeskoku przez pozycje startowe** – możliwość wykorzystywania podczas wykonywania wieloskoku własnych pozycji startowych. Łatwo można zauważać, że ta właściwość ma znaczenie jedynie podczas możliwości wykonywania wieloskoków oraz braku możliwości powrotu na własne pozycje startowe.

Ponadto wszystkie wersje gry nie pozwalają na wykonanie „pustego posunięcia”¹, czyli inaczej rezygnacji z ruchu. Dotyczy to szczególnie wariantów D i F w których możliwość wykonywania wieloskoku oraz ponownego zajmowania pozycji startowych pozwala na wykonanie takiego pustego posunięcia.

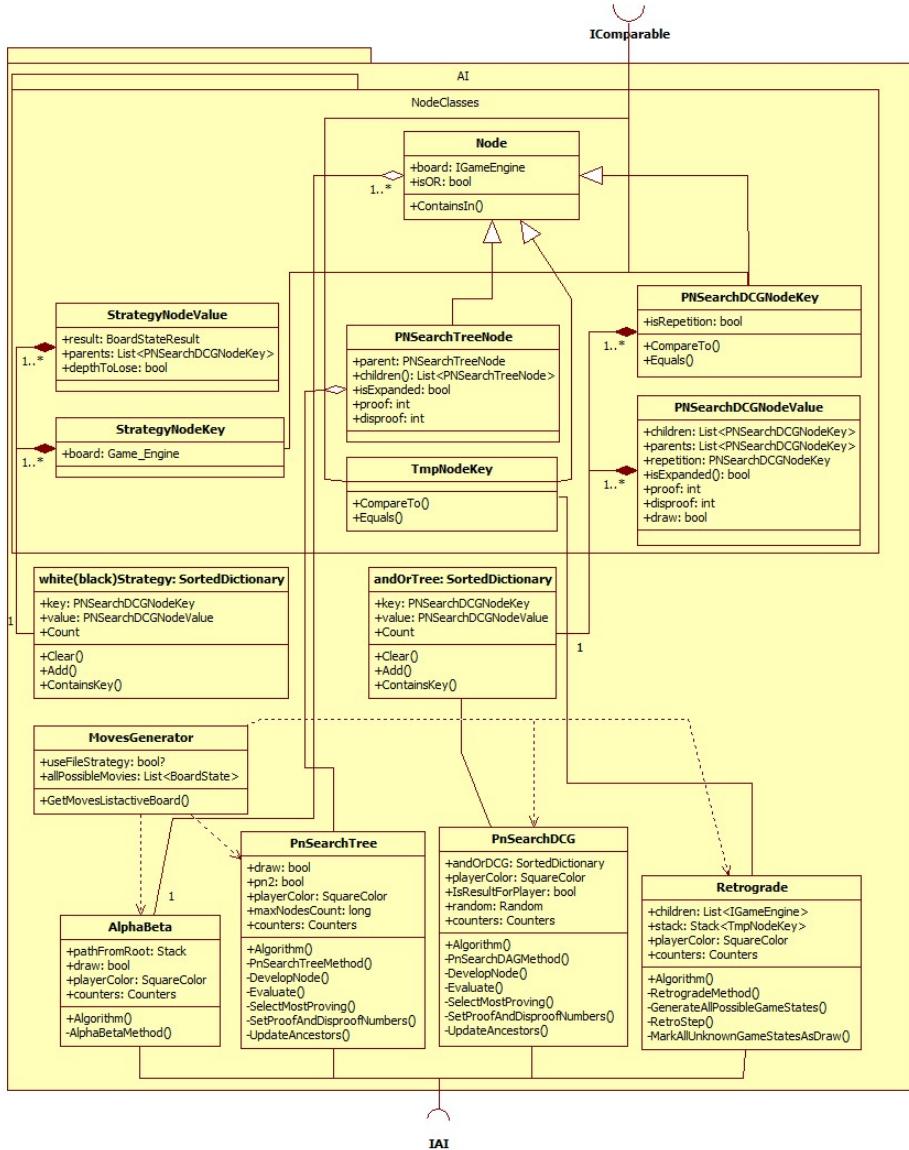
5.6. Pakiet algorytmów rozwiązywających – moduł sztucznej inteligencji

Szczegółowy podział modułu sztucznej inteligencji na klasy został przedstawiony na rys. 5.9.

Pakiet *Weles.AI.Nodes* zawiera klasy reprezentujące węzły struktur, z którymi współpracują algorytmy rozwiązyjące. Przeważnie są to drzewa albo grafy cykliczne. Jedynie algorytm α - β do zapamiętywania aktualnie przebytej ścieżki wykorzystuje stos. Struktury drzewiaste są reprezentowane za pomocą klas posiadających wskaźniki na swoje węzły potomne oraz na przodka. Grafy, ze względu na konieczność sprawdzania czy dodawany węzeł już w nim nie występuje, wymagają użycia kolekcji pozwalających na szybkie wyszukiwanie elementów. Takie właściwości spełniają kolekcje *SortedDictionary(TKey, TValue)* standardowo dostępne w środowisku .NET. Klasy, które są w nich wykorzystywane jako klucze *TKey* muszą dziedziczyć interfejs *IComparable*.

Klasa *MovesGenerator* ma za zadanie ujednolicenie działania omawianego pakietu. Generuje ona za pomocą wybranego algorytmu i konkretnego silnika gry listę możliwych posunięć wraz ich oceną z punktu widzenia aktywnego gracza.

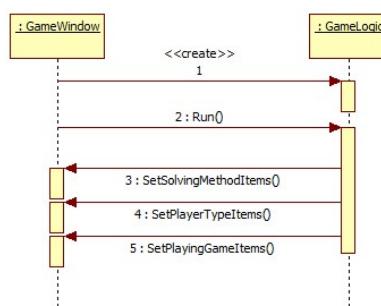
¹Pod pojęciem „pustego posunięcia” rozumie się posunięcie wg trywialnej strategii remisowej dokładnie opisane w sekcji 4.3.



RYSUNEK 5.9. Diagram klas modułu algorytmów rozwiązywających

5.7. Aspekty dynamiczne programu

W tej sekcji przedstawiono, wykorzystując diagramy sekwencji, zachowanie się programu podczas przeprowadzania rozgrywki człowieka z komputerem. Aby rozpocząć pracę z programem należy uruchomić plik wykonywalny „WelesGUI.exe”. Inicjalizacja graficznego interfejsu użytkownika została zobrazowana na rys. 5.10. W pierwszym kroku powoływana jest instancja klasy *GameLogic*, a następnie przekazywane jest do niej sterowanie poprzez wykonanie metody *Run()*. Obiekt klasy *Game-*

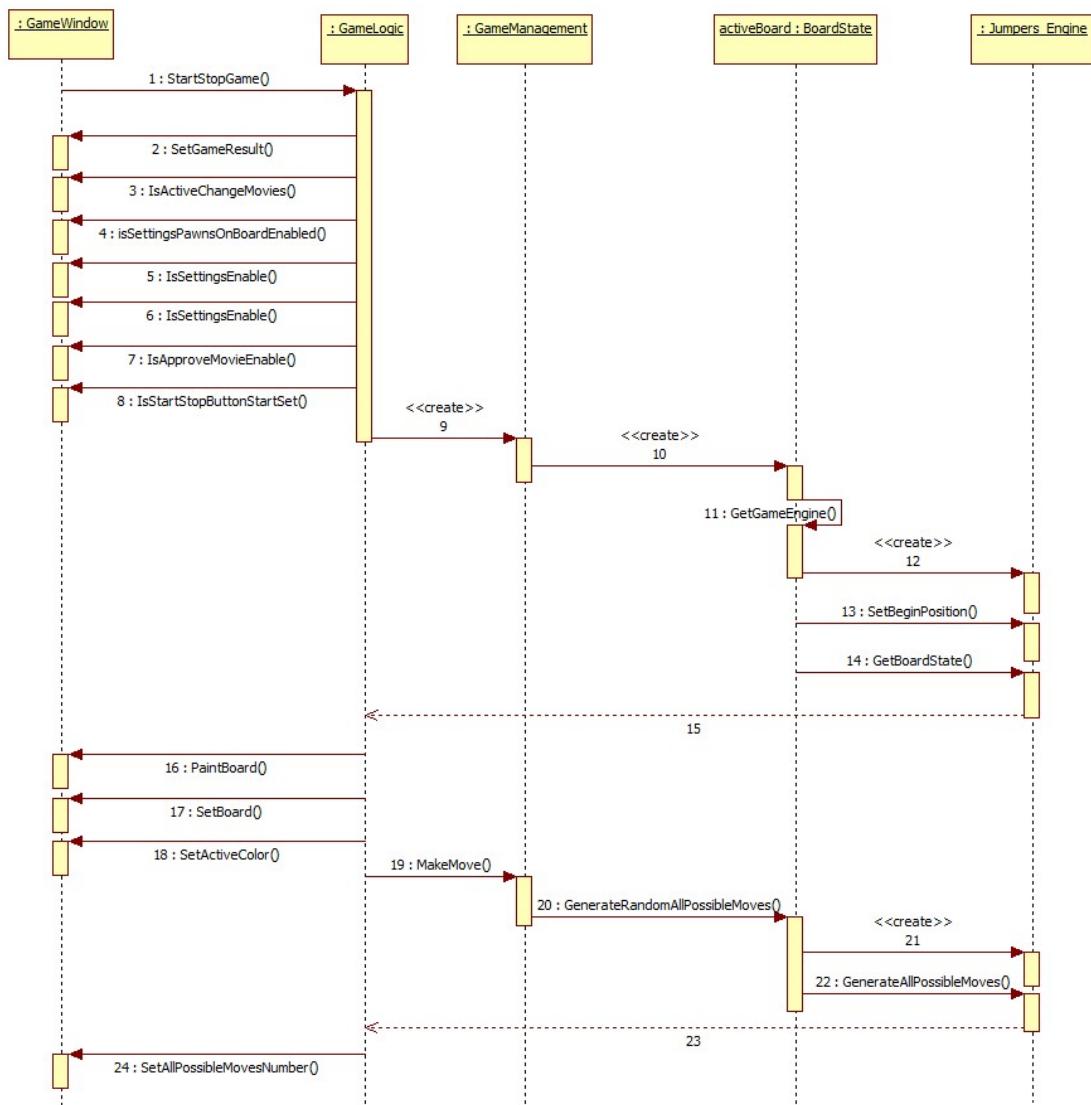


RYSUNEK 5.10. Diagram Sekwencji – Inicjalizacja GUI

Logic po przejęciu sterowania wywołuje kolejno metody *SetSolvingMethodItems()*, *SetPlayerTypeItems()* oraz *SetPlayingGameItems()*. Powoduje to wypełnienie list trzech następujących kontrolek: kontrolka algorytmów rozwiązywających grę, możliwych rodzajów graczy i możliwych wariantów gry. Informacje te są przechowywane w dwóch typach wyliczeniowych: *PlayerType* oraz *PlayingGame*. Kontrolki algorytmów rozwiązywających grę i możliwych rodzajów graczy są wypełniane identycznymi danymi z takim wyjątkiem, że algorytmem rozwiązym grę nie może być „człowiek” i „algorytm losowy”, a mogą być one typem gracza.

Po wykonaniu powyższych metod program czeka na reakcję użytkownika, który dokonuje odpowiednich ustawień polegających na wyborze pożądanych rozmiarów planszy, silnika gry oraz rodzajów graczy. Po dokonaniu tych ustawień użytkownik musi nacisnąć przycisk z etykietą „Graj” aby rozpocząć rozgrywkę.

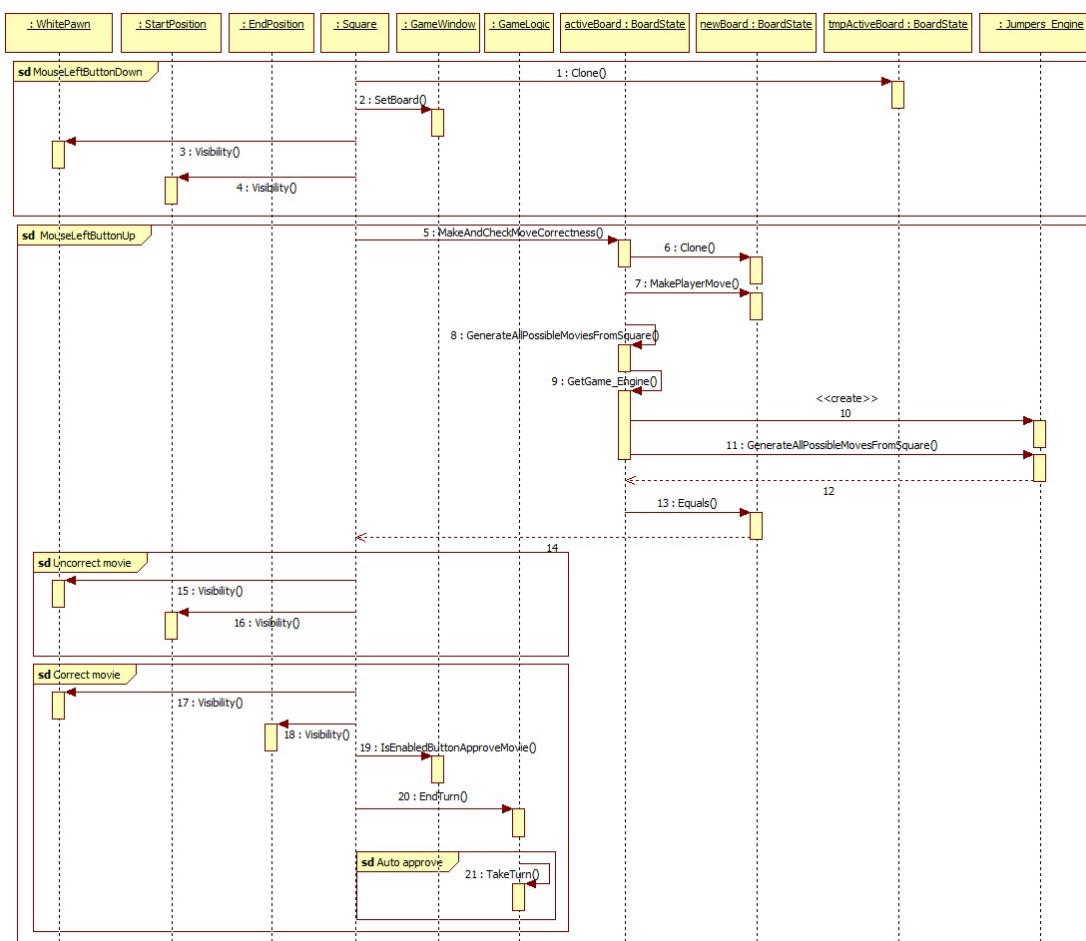
Scenariusz, który opisujemy będzie dotyczył gry człowieka z dowolnym algorytmem rozwiązymającym, przy założeniu że człowiek wykonuje pierwsze posunięcie (białym). Po naciśnięciu przycisku uruchamiającego rozgrywkę następuje wywołanie, z obiektu klasy *GameWindow*, metody *StartStopGame()* obiektu *GameLogic* – rys. 5.11. Obiekt logiki dokonuje wstępnych ustawień w interfejsie użytkownika pozwalających na rozpoczęcie gry. Polegają one na aktywacji bądź dezaktywacji pewnych grup kontrolek które są bądź nie wymagane podczas rozgrywki. Następnie powołana zostaje instancja klasy *GameManagement*, która to powołuje instancję klasy *BoardState* reprezentującą aktualny stan planszy do gry. Obiekt planszy podczas inicjalizacji wykonuje metodę *GetGameEngine()*, która tworzy obiekt silnika gry zgodnie z wcześniejszym wyborem gracza, a następnie ustawia go w stan początkowy *SetBeginPosition()* i pobiera te ustawienia ze zwracanego przez niego obiektu poprzez wywołanie metody *GetBoardState()*. Po tych operacjach obiekt odpowiadający za aktualną planszę do gry reprezentuje początkową pozycję na planszy, można zatem narysować tę planszę w GUI użytkownika. Dokonuje tego obiekt logiki poprzez



RYSUNEK 5.11. Diagram Sekwencji – start rozgrywki

wykonanie metod `PaintBoard()`, `SetBoard()` i `SetActiveColor()`. Teraz, jeśli posunięcie należało by do komputera, to zostałaby automatycznie wywołana metoda `TakeTurn()` jednak posunięcie należy do człowieka. Zostaje zatem wywołana metoda `MakeMove()`, która poprzez nowo wywołany obiekt silnika gry tworzy listę możliwych posunięć, a następnie wyświetla ich liczbę w interfejsie użytkownika za pomocą instrukcji `SetAllPossibleMovesNumber()`. W tym momencie sterowanie zostaje przekazane do GUI oczekując na ruch gracza (człowieka).

Na rys. 5.12 przedstawiono zachowanie się programu po wybraniu przez gracza poprawnego jak i nie poprawnego posunięcia. Wykonanie posunięcia składa się



RYSUNEK 5.12. Diagram Sekwencji – ruch człowieka (białymi)

z dwóch zdarzeń: Przyciśnięcie lewego przycisku myszy na polu (obiekcie) klasy Square oraz na jego zwolnieniu nad innym polem tej samej klasy. Pierwsze zdarzenie jest oznaczone ramką z etykietą: „**sd MouseLeftButtonDown**” i zaczyna działanie od przywrócenia stanu planszy przechowywanego w obiekcie `tmpActiveBoard` oraz jego narysowanie w GUI za pomocą metody `SetBoard()`. Powyższa operacja jest wykonywana dlatego, że plansza może zawierać pozycję z już wykonanym posunięciem przez gracza, ale jeszcze nie zaakceptowanym i brak takiego zabezpieczenia pozwalałby na wykonywanie przez gracza ciągu posunięć w jednej turze. Następnie, jeżeli pole na którym kliknięto zawiera pionek gracza (w tym przypadku biały), zostają wykonane dwie metody `Visibility`, w rezultacie ich działania pionek na klikniętym polu przestaje być widoczny, a aktywowany jest znacznik informujący, że stąd zostało wykonane posunięcie.

Pierwszym zadaniem metody realizującej zdarzenie zwolnienia lewego przycisku

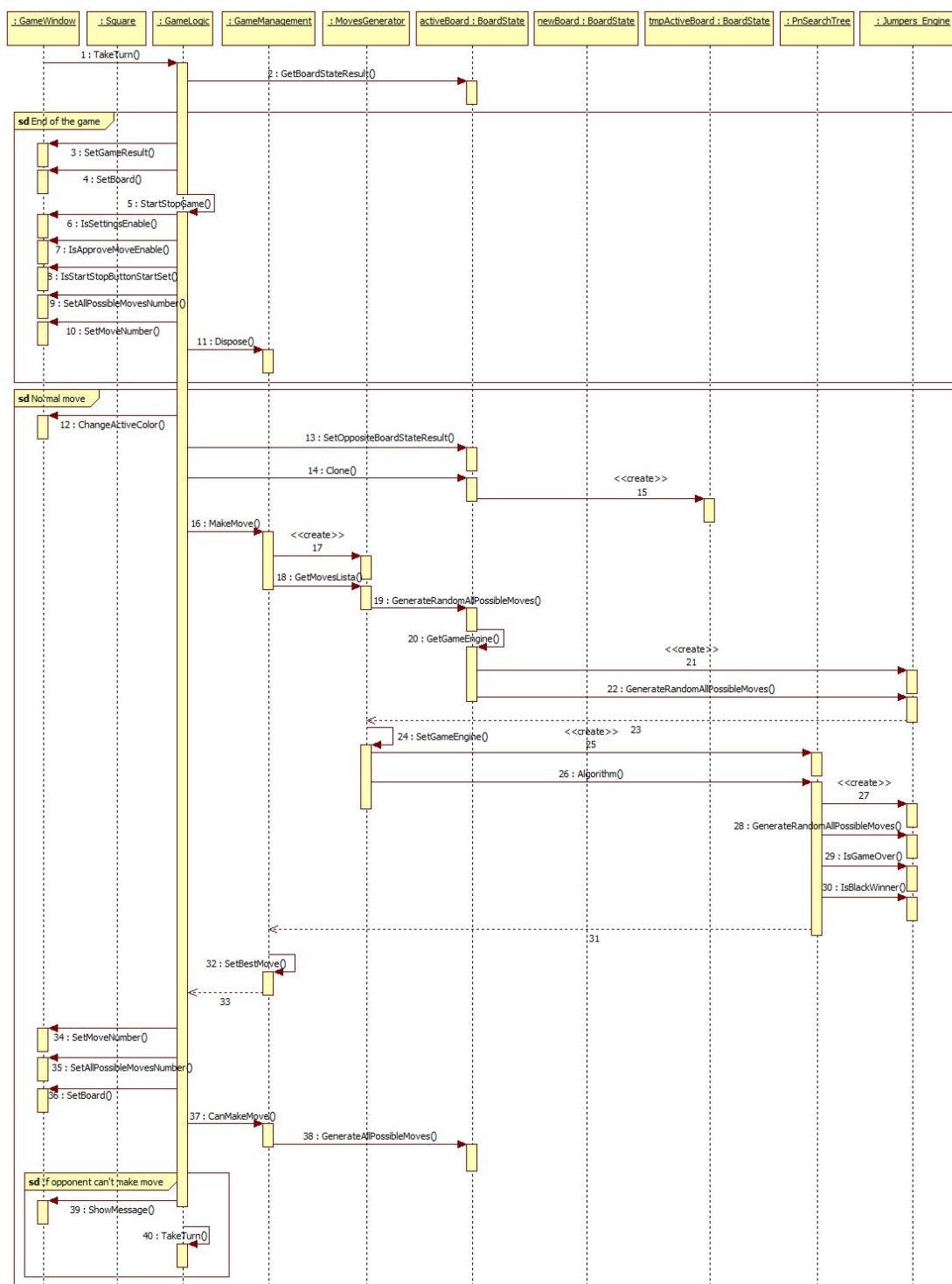
myszy zamieszczonej w ramce z etykietą: „**sd** MouseLeftButtonDown” jest sprawdzenie poprawności wykonanego posunięcia przez człowieka. Sprawdzenie to wykonywane jest przy wykorzystaniu funkcji *MakeAndCheckMoveCorrectness()*, która od razu wprowadza zmiany w obiekcie *tmpActiveBoard*, oczywiście jeśli posunięcie jest poprawne. Sprawdzenie to polega na skłonowaniu obiektu *activeBoard* pod nazwą *newBoard* i wprowadzeniu do niego zmian obrazujących posunięcie gracza. Następnie zostaje wygenerowana lista wszystkich możliwych posunięć i wszystkie z nich są porównywane z nowym obiektem. Jeżeli któreś z nich jest identyczne to posunięcie jest poprawne, w innym przypadku posunięcie jest niepoprawne.

Jeżeli przycisk myszy zostanie zwolniony na polu niezgodnym z zasadami gry (ramka z etykietą: „**sd** Uncorrect movie”), pionek zostaje odstawiony na pole na którym się znajdował, a pole to przestaje być oznaczone jako pole skąd wykonano posunięcie.

Jeżeli przycisk myszy zostanie zwolniony na polu zgodnym z zasadami gry, wykonany zostanie wariant przedstawiony w ramce z etykietą: „**sd** Correct movie”. Zostaną wykonane dwie metody *Visibility*, w których rezultacie pionek na upuszczonym polu staje się widoczny i staje się ono również widoczne jako pole, na którym zakończono posunięcie. Następnie przycisk pozwalający na akceptację tego posunięcia zostaje uaktywniony i wywołana zostaje funkcja *EndTurn()*, która sprawdza czy jest aktywne auto-zatwierdzanie posunięcia, jeśli tak to akceptacja posunięcia następuje automatycznie poprzez wywołanie funkcji *TakeTurn()*.

Po zaakceptowaniu przez gracza jego posunięcia (może to nastąpić na dwa sposoby poprzez naciśnięcie przycisku z etykietą „Zatwierdź ruch” bądź jak wspomniano wcześniej po wykonaniu poprawnego posunięcia i aktywnej funkcji autoakceptacji) następuje posunięcie przeciwnika. W rozpatrywanym przez nas przypadku jest to gracz komputerowy. Odpowiadający temu posunięciu diagram sekwencji został przedstawiony na rys. 5.13. Wywołanie tego posunięcia, jak wcześniej wspomniano, polega na wywołaniu metody *TakeTurn()*.

Na początku następuje sprawdzenie czy wcześniejszy ruch nie był posunięciem kończącym (czyli wygrywającym dla tego kto go wykonał). Decyzja ta jest podejmowana na podstawie wyniku działania metody *GetBoardStateResult()*. Jeśli reprezentuje on wygraną, to następuje koniec rozgrywki, który został przedstawiony w ramce z etykietą: „**sd** End of the game”. W pierwszym kroku zostaje uaktywniona etykieta wyświetlająca gracza wygrywającego – metoda *SetGameResult()*. Wykonanie instrukcji *SetBoard()* ma znaczenie kosmetyczne i powoduje oczyszczenie planszy ze znaczników pól początkowego i końcowego ostatniego posunięcia. Następnie wykonana zostaje metoda *StartStopGame()*, która może również zostać wykonana przez



RYSUNEK 5.13. Diagram Sekwencji – ruch komputera

gracza za pomocą naciśnięcia przycisku z etykietą "Zakończ". Powoduje ona ak-

tywację bądź dezaktywację pewnych grup kontrolek, które są bądź nie wymagane przed rozpoczęciem rozgrywki oraz poprawne zapisanie pliku buforującego wyniki algorytmu rozwiązującego.

W ramce z etykietą: „**sd** Normal movie” zamieszczono przebieg „normalnego” ruchu gracza komputerowego. W pierwszym kroku zmieniony zostaje stan kontrolki pokazującej aktywnego gracza na przeciwny. Następnie zostaje zmieniony status rozgrywki w obiekcie *activeBoard*. Status rozgrywki jest ustalany przez algorytm rozwiązujący i oznacza czy dany stan jest dla gracza aktywnego wygrywający, przegrywający czy remisowy. Następnie na obiekcie *activeBoard* zostaje wywołana funkcja *Clone()*, w wyniku jej działania zostaje powołany obiekt *tmpActiveBoard*, który może zostać wykorzystany do przywrócenia aktualnego stanu planszy. Kolejną metodą jest *MakeMove* i jest ona odpowiedzialna za powołanie odpowiedniego algorytmu rozwiązującego oraz za ocenienie za jego pomocą wszystkich możliwych posunięć gracza komputerowego. Algorytm, o którym mowa jest reprezentowany przez metodę *Algorithm()* i wykorzystuje do swojego działanie trzy następujące funkcje silnika gry: *GenerateRandomAllPossibleMoves()*, *IsGameOver()* oraz *IsBlackWinner()*. Po wykonaniu metody *MakeMove* następuje wybranie najlepszego ze zwróconych posunięć (instrukcja *SetBestMove()*) oraz wyświetlenie go na planszy, wraz z liczbą możliwych posunięć i jego numeru. Na końcu następuje sprawdzenie, czy przeciwnik nie został przypadkiem zablokowany, jeśli tak to wykonywany jest wariant przedstawiany w ramce zaetykietowanej „**sd** If opponent can't make movie”. Polega on na wyświetleniu stosownej informacji i automatycznym wykonaniu posunięcia.

Powyższe diagramy pokazują większość najistotniejszych zachowań się programu. Dają wgląd w jego działanie oraz logikę współpracy poszczególnych bloków. Ze względu na ograniczenia przestrzenne pracy oraz z racji dużego podobieństwa w zachowaniu w przypadku realizacji innych funkcji programu autor zdecydował się zrezygnować z ich zamieszczenia.

Rozdział 6

Wyniki eksperymentu obliczeniowego

6.1. Własności gry *Skoczki*

W pierwszym kroku oszacujemy przestrzeń stanów rozwiązywanej gry. Wiemy, że liczba pionków nie zmienia się podczas trwania rozgrywki. Wynika z tego fakt, że poprawne są jedynie stany w których: dwanaście pól będzie w stanie *zajęte przez biały pionek*, dwanaście w stanie *zajęte przez czarny pionek*, a pozostałe czterdzieści dwa pola będą w stanie *wolne*. Zatem, aby dowiedzieć się czegoś o grze można oszacować rozmiar jej przestrzeni stanów. W tym celu wykorzystamy przytoczony w sekcji 3.1.1 wzór na liczbę wszystkich permutacji z uwzględnieniem powtórzeń.

$$P_{n_1, n_2, \dots, n_k} = \frac{(n_1 + n_2 + \dots + n_k)!}{(n_1)! \cdot (n_2)! \cdot \dots \cdot (n_k)!} \quad (6.1)$$

Górne oszacowanie rozmiaru przestrzeni stanów dla planszy o oryginalnych rozmiarach (6×11) wg Piąnowskiego [40] wynosi:

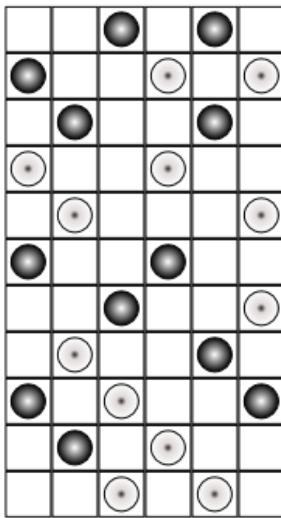
$$P_{12,12,42} = \frac{66!}{12! \cdot 12! \cdot 42!} \approx 1,69 \cdot 10^{24} \quad (6.2)$$

Rozpatrując planszę o rozmiarach (8×8) reprezentującą inny, komercyjny wariant gry opisany w sekcji 4.4, rozważane ograniczenie wynosi:

$$P_{16,16,32} = \frac{64!}{16! \cdot 16! \cdot 32!} \approx 1,1 \cdot 10^{27} \quad (6.3)$$

Jak widać, przestrzeń stanów już jest olbrzymia! Dodatkowo, z powodu wymaganej informacji o tym który z graczy wykonuje aktualnie posunięcie ulegnie ona podwojeniu. Wiemy jednak, że wiele stanów w grze jest nieosiągalnych. Liczba tych stanów mocno zależy od rozważanego wariantu gry. Warianty te opisano w sekcji 5.5 w tab. 5.1. Przykładowo, dla zasad gry, w których niedozwolone są pojedyncze posunięcia (A, B-1, B-2, C, D) stan, w którym żadne dwa pionki nie stykają się bokami jest nieosiągalny. Powodowałby on również zakończenie gry z powodu braku możliwości wykonania jakiegokolwiek posunięcia przez któregoś z graczy. Przykład takiego niedozwolonego stanu przedstawiono na rys. 6.1.

Jak już wcześniej wspomniano, omawiana gra jest łatwo skalowalna. Spróbujmy zatem określić górne ograniczenie na rozmiar przestrzeni stanów w zależności od



RYSUNEK 6.1. Nieosiągalna pozycja w każdym wariantie gry *Skoczki* pozwalającym jedynie na skoki

rozmiarów jej planszy. Taką zależność można opisać za pomocą następującego wzoru wzoru:

$$P_{n_b, n_c, m} = \frac{(n_b + n_c + m)!}{n_b! \cdot n_c! \cdot m!} \quad (6.4)$$

gdzie:

- n_b – liczba białych pionków
- n_c – liczba czarnych pionków
- m – liczba wolnych pól

spełniających warunek:

- $n_b + n_c + m = w * s$

gdzie:

- w – wysokość planszy
- s – szerokość planszy

Wyniki szacowań górnych ograniczeń przestrzeni stanów przedstawiono w tab. 6.1. Pogrubioną czcionką oznaczono rozmiary planszy dla dwóch wcześniejszych oszacowanych wariantów tej gry. Zamieszczone wyniki zostały pomnożone przez 2 w celu uwzględnienia informacji o graczu, do którego należy posunięcie w danym stanie gry.

TABLICA 6.1. Górnne oszacowanie rozmiaru przestrzeni stanów gry *Skoczki* o różnych wymiarach

	1	2	3	4	5	6	7	8
5	60	6300	840840	124710300	19632172560	$3,2113 \cdot 10^{12}$	$5,3953 \cdot 10^{14}$	$9,2452 \cdot 10^{16}$
6	180	69300	34306272	18931023540	$1,1102 \cdot 10^{13}$	$6,7695 \cdot 10^{15}$	$4,2411 \cdot 10^{18}$	$2,7107 \cdot 10^{21}$
7	420	420420	543182640	$7,8306 \cdot 10^{11}$	$1,2002 \cdot 10^{15}$	$1,9129 \cdot 10^{18}$	$3,1331 \cdot 10^{21}$	$5,2354 \cdot 10^{24}$
8	840	1801800	4997280288	$1,5472 \cdot 10^{13}$	$5,0936 \cdot 10^{16}$	$1,7441 \cdot 10^{20}$	$6,1368 \cdot 10^{23}$	$2,2031 \cdot 10^{27}$
9	1512	6126120	32125373280	$1,8810 \cdot 10^{14}$	$1,1713 \cdot 10^{18}$	$7,5860 \cdot 10^{21}$	$5,0491 \cdot 10^{25}$	$3,4288 \cdot 10^{29}$
10	2520	17635800	$1,5984 \cdot 10^{11}$	$1,6178 \cdot 10^{15}$	$1,7415 \cdot 10^{19}$	$1,9498 \cdot 10^{23}$	$2,2436 \cdot 10^{27}$	$2,6340 \cdot 10^{31}$
11	3960	44767800	$6,5570 \cdot 10^{11}$	$1,0726 \cdot 10^{16}$	$1,8662 \cdot 10^{20}$	$3,3772 \cdot 10^{24}$	$6,2809 \cdot 10^{28}$	$1,1919 \cdot 10^{33}$

Widać, że dla mniejszych rozmiarów planszy przestrzeń stanów gry jest niewielka i pierwszym doświadczeniem, które należało by przeprowadzić, jest wygenerowanie wszystkich stanów występujących w poszczególnych wariantach gry, a następnie porównanie tych wielkości z naszymi oszacowaniami. Do tego eksperymentu najlepiej nadaje się metoda *GenerateAllPossibleGameStates(root)* która jest częścią analizy wstecznej opisanej szczegółowo w sekcji 3.2. Wyniki tych porównań przedstawiono w tab. 6.2. Zamieszczono w niej wartości bezwzględne zarówno pomiarów jak i oszacowań. Oczywiście uwzględniono jedynie te rozmiary plansz dla których udało się zmieścić w pamięci operacyjnej wszystkie stany gry. W tab. 6.3 przedstawiono te same dane ale w postaci procentowej, określającej stosunek rzeczywistej przestrzeni możliwych do osiągnięcia stanów gry z prezentowanymi wcześniej, górnymi oszacowaniami na tę przestrzeń.

Zgodnie z naszymi wcześniejszymi oczekiwaniami rzeczywista przestrzeń stanów gry, które mogą w niej wystąpić jest w dużym stopniu uzależniona od przyjętych reguł gry. Poszczególne warianty gry celowo zostały zamieszczone w takiej kolejności. Począwszy od najbardziej restrykcyjnych, pozostawiających graczu jak najmniejszą liczbę możliwych posunięć do takich, w których współczynnik rozgałęzienia drzewa gry jest największy. Dla wariantu A, w którym można wykonywać tylko pojedyncze przeskoki i nie wolno powracać na pozycje początkowe stosunek pomiędzy rzeczywistą przestrzenią stanów gry a górnym oszacowaniem wynosił 22,73% (wartość najwyższa) i malał wraz ze wzrostem rozmiarów planszy aż do 0,00049%. Można przypuszczać więc, że dla jeszcze większych rozmiarów planszy będzie on jeszcze mniejszy. Natomiast dla wariantów E i F w którym gracz może wykonywać zarówno pojedyncze posunięcia na dowolne sąsiednie wolne pole

TABLICA 6.2. Porównanie oszacowania rozmiaru przestrzeni stanów gry *Skoczki* z rzeczywistymi wartościami

rozm.	rozmiar przestrzeni stanów:						oszacowany
	A	B2	B1	C	D	E i F	
1×5	8	8	8	11	11	54	60
1×6	6	11	11	152	158	168	180
1×7	18	22	22	358	376	400	420
1×8	108	108	108	722	736	810	840
1×9	250	250	250	1.248	1.266	1.470	1.512
1×10	502	502	502	1.990	2.012	2.464	2.520
1×11	900	900	900	2.976	3.002	3.888	3.960
2×5	155	155	155	1.782	1.782	6.270	6.300
2×6	510	907	907	16.128	16.128	69.160	69.300
2×7	4.296	5.958	5.958	83.032	83.032	420.000	420.420
2×8	27.432	27.446	27.446	303.750	303.750	1.800.810	1.801.800
2×9	113.394	113.394	113.394	886.230	886.230	—	6.126.120
2×10	351.266	351.266	351.266	2.203.432	2.203.432	—	17.635.800
2×11	969.060	969.060	969.060	4.864.608	4.864.608	—	44.767.800
3×5	2.530	5.461	5.987	188.910	188.910	840.672	840.840
3×6	141.716	368.863	414.946	—	—	—	34.306.272
3×7	1.176.576	8.595.936	8.410.830	—	—	—	543.182.640
4×5	59.552	1.781.852	3.991.600	—	—	—	124.710.300
5×5	962.180	—	—	—	—	—	19.632.172.560
6×5	15.767.421	—	—	—	—	—	3.211.320.457.800

jak i zajmować pozycje startowe, nasze szacowanie jest bardzo bliskie rzeczywistemu rozmiarowi przestrzeni stanów. Wacha się ono od 90% (wartość najmniejsza) i rośnie wraz ze wzrostem rozmiarów planszy aż do 99,98%.

Kolejnym eksperymentem, który przeprowadzono było porównanie wydajności poszczególnych algorytmów. Porównanie to dotyczyło przede wszystkim wielkości instancji problemu, z którym sobie dany algorytm poradził. Dodatkowo były porównywane:

- czasy wykonania;
- liczba wszystkich rozwiniętych węzłów;
- maksymalna liczba węzłów przechowywana jednocześnie w pamięci.

TABLICA 6.3. Porównanie oszacowania rozmiaru przestrzeni stanów gry *Skoczki* z rzeczywistymi wartościami (wartości względne)

rozm.	rozmiar przestrzeni stanów: rzeczywisty dla wariantu gry:					
	A	B2	B1	C	D	E i F
2×5	13,33%	13,33%	13,33%	18,33%	18,33%	90%
2×6	3,33%	6,11%	6,11%	84,44%	87,78%	93,33%
2×7	4,29%	5,24%	5,24%	85,24%	89,52%	95,24%
2×8	12,86%	12,86%	12,86%	86%	87,62%	96,43%
2×9	16,53%	16,53%	16,53%	82,54%	83,73%	97,22%
2×10	19,92%	19,92%	19,92%	79%	79,84%	97,78%
2×11	22,73%	22,73%	22,73%	75,15%	75,81%	98,18%
2×5	2,46%	2,46%	2,46%	28,29%	28,29%	99,52%
2×6	0,74%	1,31%	1,31%	23,27%	23,27%	99,8%
2×7	1,02%	1,42%	1,42%	19,75%	19,75%	99,9%
2×8	1,52%	1,52%	1,52%	16,86%	16,86%	99,95%
2×9	1,85%	1,85%	1,85%	14,47%	14,47%	—
2×10	1,99%	1,99%	1,99%	12,49%	12,49%	—
2×11	2,16%	2,16%	2,16%	—	—	—
3×5	0,3%	0,65%	0,71%	22,47%	22,47%	99,98%
3×6	0,41%	1,08%	1,21%	—	—	—
3×7	0,22%	1,58%	1,55%	—	—	—
4×5	0,048%	1,43%	3,2%	—	—	—
5×5	0,0049%	—	—	—	—	—
6×5	0,00049%	—	—	—	—	—

Porównaniu będą podlegały następujące algorytmy:

- analizy wstępnej – omówiony w sekcji 3.2

- $\alpha\text{-}\beta$ – omówiony w sekcji 3.3.3
- $pn\text{-}search^1$ współpracujący z drzewem And/Or – omówiony w sekcji 3.5.1
- $pn^2\text{-}search$ przy współpracy z drzewem And/Or – omówiony w sekcji 3.5.3
- $pn\text{-}search$ współpracujący z DCG – omówiony w sekcji 3.5.6

TABLICA 6.4. Wartość pozycji startowej gry *Skoczki* w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.1

rozmiar	wynik	retrograde	czas trwania obliczeń:			
			$\alpha\text{-}\beta$	pn-search	$pn^2\text{-}search$	pn-search.DCG
1×5	przegrana	<i>0,31ms</i>	<i>0,09ms</i>	<i>0,16ms</i>	<i>0,21ms</i>	<i>0,15ms</i>
		8	5	6	6	6
		8	4	6	6	6
1×6	wygrana	<i>0,3ms</i>	<i>0,09ms</i>	<i>0,16ms</i>	<i>0,23ms</i>	<i>0,17ms</i>
		6	6	6	6	6
		6	5	6	6	6
1×7	wygrana	<i>0,87ms</i>	<i>0,25ms</i>	<i>0,35ms</i>	<i>0,47ms</i>	<i>0,56ms</i>
		18	16	11	11	14
		18	9	11	11	14
1×8	wygrana	<i>8,73ms</i>	<i>0,36ms</i>	<i>0,37ms</i>	<i>0,53ms</i>	<i>0,57ms</i>
		108	22	12	12	14
		108	13	12	12	14
1×9	przegrana	<i>29,6ms</i>	<i>0,41-1,08ms</i>	<i>0,61ms</i>	<i>0,82ms</i>	<i>1,35ms</i>
		250	24-56	19	19	24
		250	13-19	17	17	24
1×10	wygrana	<i>74,1ms</i>	<i>0,28-162ms</i>	<i>0,69ms</i>	<i>0,96ms</i>	<i>1,67ms</i>
		502	16-7.192	21	21	28
		502	13-112	19	19	28
1×11	wygrana	<i>162ms</i>	<i>1,48-52.200ms</i>	<i>3,85ms</i>	<i>6,18ms</i>	<i>7,91ms</i>
		900	75-1.914.124	108	129	81
		900	21-237	63	91	81

Algorytm analizy wstępnej jest jedynym algorytmem z wymienionych, który podczas swojego działania rozwiązuje wartość każdego stanu jaki może w niej wystąpić. Może zostać zatem wykorzystany do badania poprawności działania pozostałych al-

¹Pełna nazwa: *Proof-Number Search*.

TABLICA 6.5. Wartość pozycji startowej gry *Skoczki* w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.2

rozmiar	wynik	czas trwania obliczeń:				
		retrograde	$\alpha-\beta$	pn-search	pn ² -search	pn-search.DCG
2×5	przegrana	14,7ms 155 155	0,87-1,53ms 47-105 9-11	3,8ms 135 36	5,63ms 135 48	7,84ms 90 90
		55,4ms 510 510	9,94-19,6ms 559-1047 30-42	8,43ms 247 210	11,4ms 247 210	14,8ms 98 98
		1,43s 4.296 4.296	— — —	0,59s 11.415 5.290	0,82s 13.106 5.594	0,24s 685 384
2×8	remis	12,8s 27.432 27.432	— — —	14,4s 256.394 105.648	25,5s 391.542 136.312	1,84s 2.593 1.583
		1m 33s 113.394 113.394	— — —	3m 37s 3.443.458 1.902.129	25m 1s 25.501.854 1.062.426	3,2s 4.806 2.522
		6m 44s 351.266 351.266	— — —	— 658.557.210 3.455.220	10h 46m 1s — —	5,76s 8.133 4.128
2×11	remis	21m 14s 969.060 969.060	— — —	— — —	— — —	16,6s 15.449 9.209

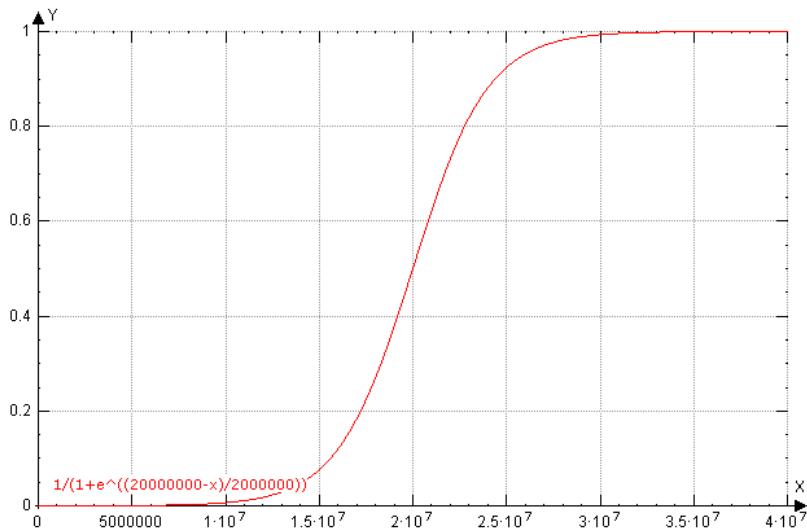
gorytmów. Jest to szczególnie wskazane w przypadku algorytmu pn-search współpracującego z DCG, który jak wiemy może „przeoczyć” wygraną któregoś z graczy. Innymi słowy może sytuację wygrywającą bądź przegrywającą dla jednego z graczy potraktować jako sytuację remisową. Wszystkie pozostałe algorytmy podczas jednego swojego przebiegu¹ są w stanie określić tylko wartość pozycji, która została im przekazana jako korzeń². W tab. 6.4, 6.5 i 6.6 zamieszczono wyniki omawianego eksperymentu. Dotyczą one wariantu A, który charakteryzuje się najmniejszym

¹W rzeczywistości algorytm *pn-search* oraz wszystkie jego pochodne mogą być uruchamiane dwukrotnie w jednym przebiegu. Wynika to z faktu występowania w grze sytuacji remisowych, które mogą być jednoznacznie wykryte dopiero po uruchomieniu algorytmu z punktu widzenia gracza i jego przeciwnika.

²Przeważnie jest to pozycja startowa gry, jednak nic nie stoi na przeszkodzie aby wyliczać wartość innych jej stanów. Jest to na przykład stosowane podczas trwania rozgrywki.

współczynnikiem rozgałęzienia i powinien zapewnić największe rozmiary planszy rozwiązań gier.

Najgorsze wyniki osiągnął algorytm $\alpha-\beta$ już dla plansz o wymiarach 2×7 i 3×6 czas obliczeń był zbyt długi do praktycznych zastosowań¹. Dodatkowo widać że czas szukania rozwiązania mocno zależy od przypadku. Ponieważ kolejność posunięć generowanych z danego stanu jest losowa, różnice w czasie wykonania się algorytmu mogą być znaczące. Widać to na przykładzie instancji problemu o rozmiarach 1×11 , gdzie w jednym przebiegu rozwiązanie zostało znalezione po odwiedzeniu 75 stanów gry, a w innym przypadku (kiedy algorytm poszedł w przysłowiowe „krzaki”) liczba ta wyniosła 1.914.124.



RYSUNEK 6.2. Wykres funkcji współczynnika regulującego liczbę wierzchołków drzewa poziomu drugiego.

Znacznie stabilniej zachowywał się algorytm *pn-search* uruchomiony na drzewie And/Or, jednak nie osiągnął on znaczących rezultatów. Instancje problemu rozwiązywanych przez niego były mniejsze, niż w przypadku najwyklejszej analizy wstępnej. Wynika to z faktu, iż pomimo wybrania wariantu A współczynnik rozgałęzienia jest zbyt wysoki w stosunku do rozmiaru przestrzeni stanów gry, a to z kolei wynika z bardzo dużej liczby powtórzeń występujących w grze, co powoduje że drzewo gry staje się olbrzymie nawet przy niewielkim rozmiarze przestrzeni stanów.

Odmiana *pn²-search* tego algorytmu, również uruchomiona na drzewie And/Or dała podobne rezultaty. Jak łatwo zauważyc, algorytm ten pozwala zaoszczędzić na wielkości wykorzystanej pamięci kosztem czasu poszukiwań. Ta właściwość ujawnia

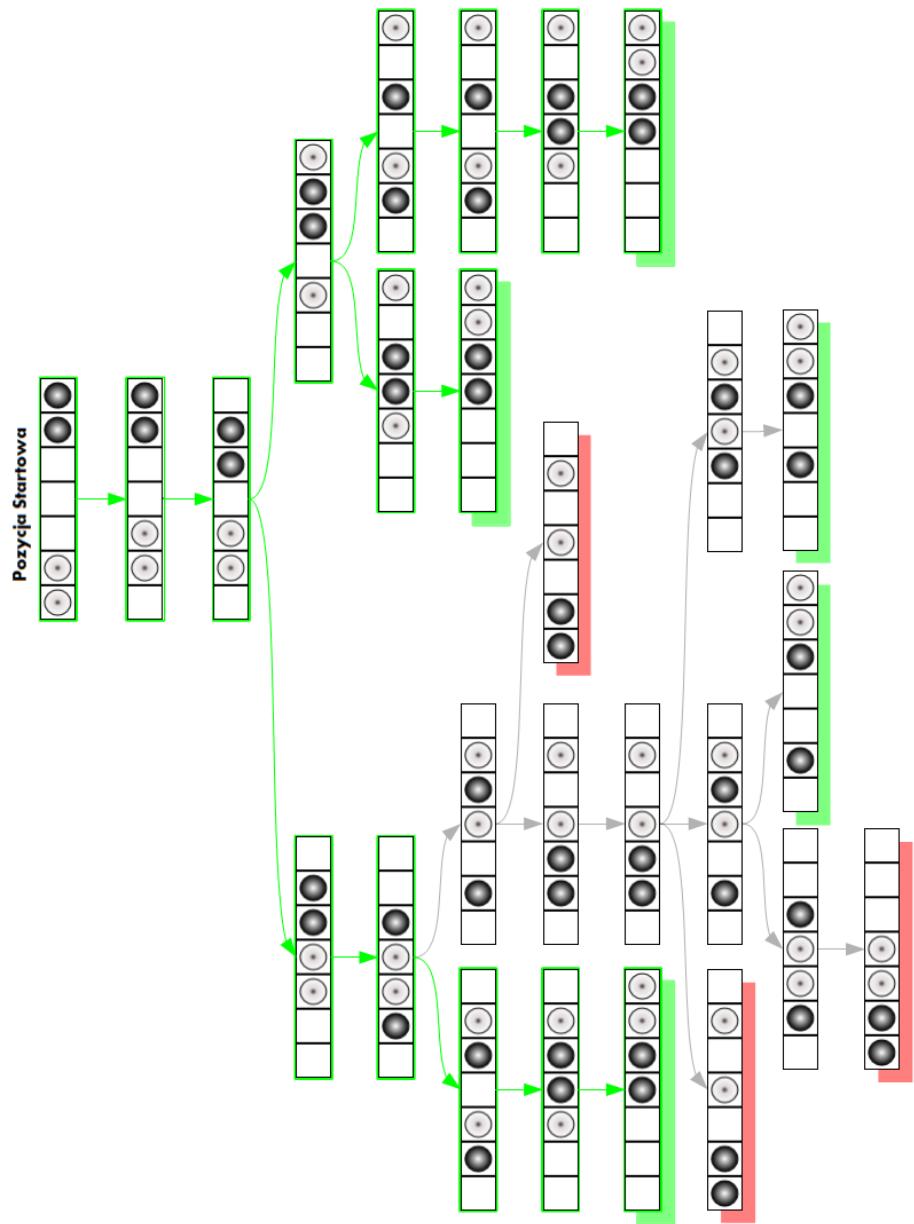
¹Algorytm nie wykonał się po tygodniu pracy.

TABLICA 6.6. Wartość pozycji startowej gry *Skoczki* w wersji A oraz czasy obliczeń i liczba rozwiniętych węzłów cz.3

rozmiar	wynik	czas trwania obliczeń:				
		retrograde	$\alpha-\beta$	pn-search	pn ² -search	pn-search_DCG
3×5	przegrana	<i>0,6s</i>	<i>41,8-181ms</i>	<i>0,35s</i>	<i>0,54s</i>	<i>0,27s</i>
		2.530	2.414-9.721	9830	10.817	1.742
		2.530	32	3450	3.435	1.742
3×6	remis	<i>1m 37s</i>	—	<i>1m 9s</i>	<i>4m 49s</i>	<i>10,1s</i>
		141.716	—	1.111.260	4.719.791	10.469
		141.716	—	621.446	525.166	5.845
3×7	remis	<i>38m 2s</i>	—	—	—	<i>2m 57s</i>
		1.176.576	—	—	—	35.519
		1.176.576	—	—	—	24.822
3×8	remis	—	—	—	—	<i>2h 0m 3s</i>
		—	—	—	—	324.549
		—	—	—	—	251.454
3×9	remis	—	—	—	—	<i>7h 37m 33s</i>
		—	—	—	—	929.666
		—	—	—	—	474.245
4×5	przegrana	<i>29s</i>	—	<i>58s</i>	<i>3m 37s</i>	<i>20,1s</i>
		59.552	—	1.330.368	4.247.653	20.986
		59.552	—	800.340	480.487	20.986
4×6	przegrana	—	—	—	—	<i>2h 14m 45s</i>
		—	—	—	—	502.536
		—	—	—	—	502.536
5×5	przegrana	<i>16m 12s</i>	—	—	—	<i>18m 51s</i>
		962.180	—	—	—	287.768
		962.180	—	—	—	287.768
6×5	przegrana	<i>9h 1m 54s</i>	—	—	—	—
		15.767.421	—	—	—	—
		15.767.421	—	—	—	—

się dopiero przy większych instancjach problemu, a to dlatego, że współczynniki a i b we wzorze 3.12 zamieszczonym w sekcji 3.5.3 zostały tak dobrane, aby największy wzrost wielkości drzewa na poziomie (pn_2) następował dopiero nieco przed rozwinięciem 20 mln węzłów na poziomie (pn_1). Wartość ta z kolei jest tak dobrana, aby wykorzystywać połowę dostępnej pamięci komputera. Pozwala to na szybki rozwój drzewa na pierwszym poziomie, a następnie jego przyhamowanie i oszczędne rozwijanie kolejnych węzłów za pomocą dużych drzew drugiego poziomu, dających lepsze

oszacowania wartości liczb dowodzących i obalających. Wykres funkcji współczynnika liczby wierzchołków drzewa na poziomie (pn_2) przedstawiono na rys. 6.2.



RYSUNEK 6.3. Drzewo gry dla planszy o wymiarach (1×7)

Najlepsze rezultaty osiągnął algorytm *pn-search* uruchomiony na DCG. Pozwolił on na ustalenie wartości pozycji startowej instancji gry o wymiarach większej, niż algorytm analizy wstępnej. Niestety wynik ten nie może być traktowany ze 100 procentową pewnością ponieważ, jak pokazano w sekcji 3.5.6, algorytm może podać

błędny wynik. Dokładne pomiary wykazały, że na planszy o wymiarach (2×8) na 27.432 możliwe stany gry 8 zostało zinterpretowanych błędnie. Zatem 0,029% pozycji zostało niewłaściwie ocenionych. Analogicznie dla planszy (2×9) na 113.394 możliwe stany, 283 zostały błędne obliczone, co stanowi już 0,25% całej przestrzeni stanów. Natomiast dla planszy (3×6) gdzie przestrzeń stanów wynosi 141.716 pozycji algorytm wykazał aż 5065 błędnych interpretacji. W tym przypadku jest to już 3,57% i wyraźnie świadczy o jego niewiarygodności.

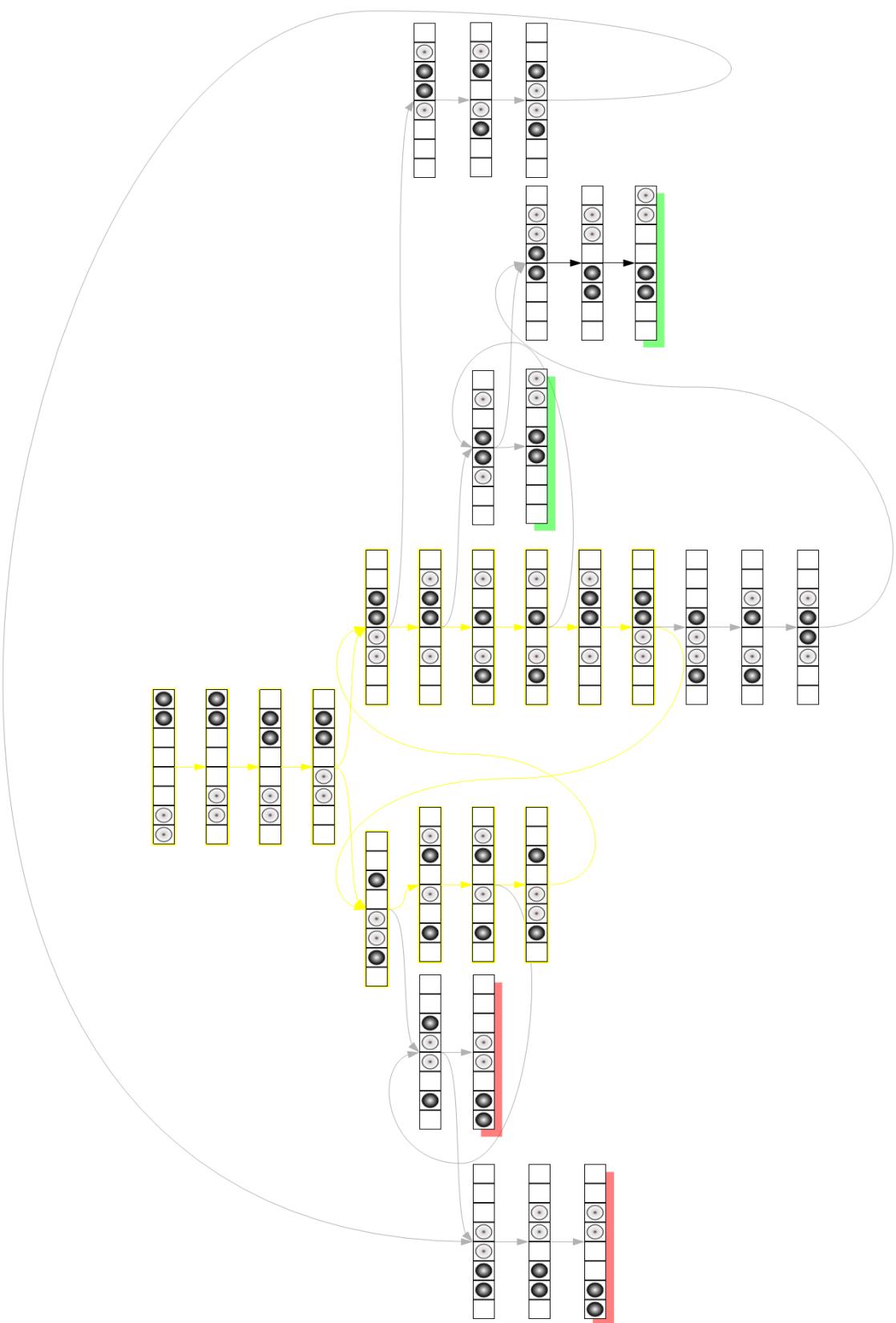
Aby przyśpieszyć rozgrywkę, szczególnie na planszach o dużych rozmiarach, program „Weles” pozwala na budowanie plików strategii dla obydwu graczy. Taki plik jest znacznie mniejszy od całkowitej przestrzeni stanów gry i posiada zawsze tylko jedno optymalne posunięcie gracza, oraz wszystkie możliwe odpowiedzi przeciwnika. Zbiór reprezentujący taką strategię został uwidoczniony na rys. 6.3¹ i przedstawia optymalną strategię wygrywającą gracza rozpoczynającego dla wariantu B.

Na rys. 6.4 przedstawiono strategie optymalne obydwu graczy dla wariantu B. Rozgrywka jest remisowa. Gracze grając optymalnie będą na przemian (w nieskończoność) wykonywali posunięcia wzdłuż żółtych łuków. Jeśli którykolwiek z nich wykona jakiekolwiek nieoptimalne posunięcie, to jego przeciwnik kontynuując optymalną strategię doprowadzi do własnej wygranej podążając wzdłuż szarych łuków.

Dla wszystkich gier, które udało się rozwiązać za pomocą algorytmu analizy wstecznej również udało się stworzyć pliki zawierające optymalną strategię obydwu graczy.

Do wszystkich obliczeń omawianych w tym rozdziale posłużył komputer przenośny (notebook) firmy BENQ, model Joybook P52 z dwurdzeniowym procesorem AMD Turion(tm) 64 X2 Mobile Technology TL-52 1,60 GHz z 2GB pamięci RAM i zainstalowanym systemem operacyjnym Windows VistaTM Business Service Pack 2.

¹Zbiór ten składa się z zielonych łuków i incydentnych do nich stanów gry.



RYSUNEK 6.4. Digraf dowodzący remis dla planszy o wymiarach 1x8

Rozdział 7

Podsumowanie

W niniejszej pracy szczegółowo omówiono kryteria podziału gier. Szczegółowo zapoznano czytelnika z terminem „rozwiązywanie gry” i jego trzema postaciami. Następnie przedstawiono historię rozwiązywania gier oraz perspektywy na przyszłość związane z tą dziedziną.

W kolejnym rozdziale szczegółowo przedstawiono podstawowe metody służące rozwiązywaniu gier. Opisano zarówno algorytmy jak i wykorzystywane przez nie struktury danych. Omówione zostały również przykłady szacowania przestrzeni stanów gry.

Następnie szczegółowo opisano zasady gry przeznaczonej do rozwiązywania i motywacje autora którymi się kierował przy jej wyborze.

W ostatnich dwóch rozdziałach opisano praktyczną część tej pracy, którą był program „Weles”. Podano jego strukturę oraz przedstawiono jego działanie podczas realizacji podstawowych funkcjonalności.

W drugim ze wspomnianych rozdziałów przedstawiono wyniki eksperymentów przeprowadzonych za pomocą wspomnianego programu. Polegały one na porównaniu skuteczności wcześniej omówionych i zaimplementowanych metod w starciu z problemem jaki stanowiły różne warianty rozwiązywanej gry oraz ustalenie wartości pozycji startowych tychże wariantów.

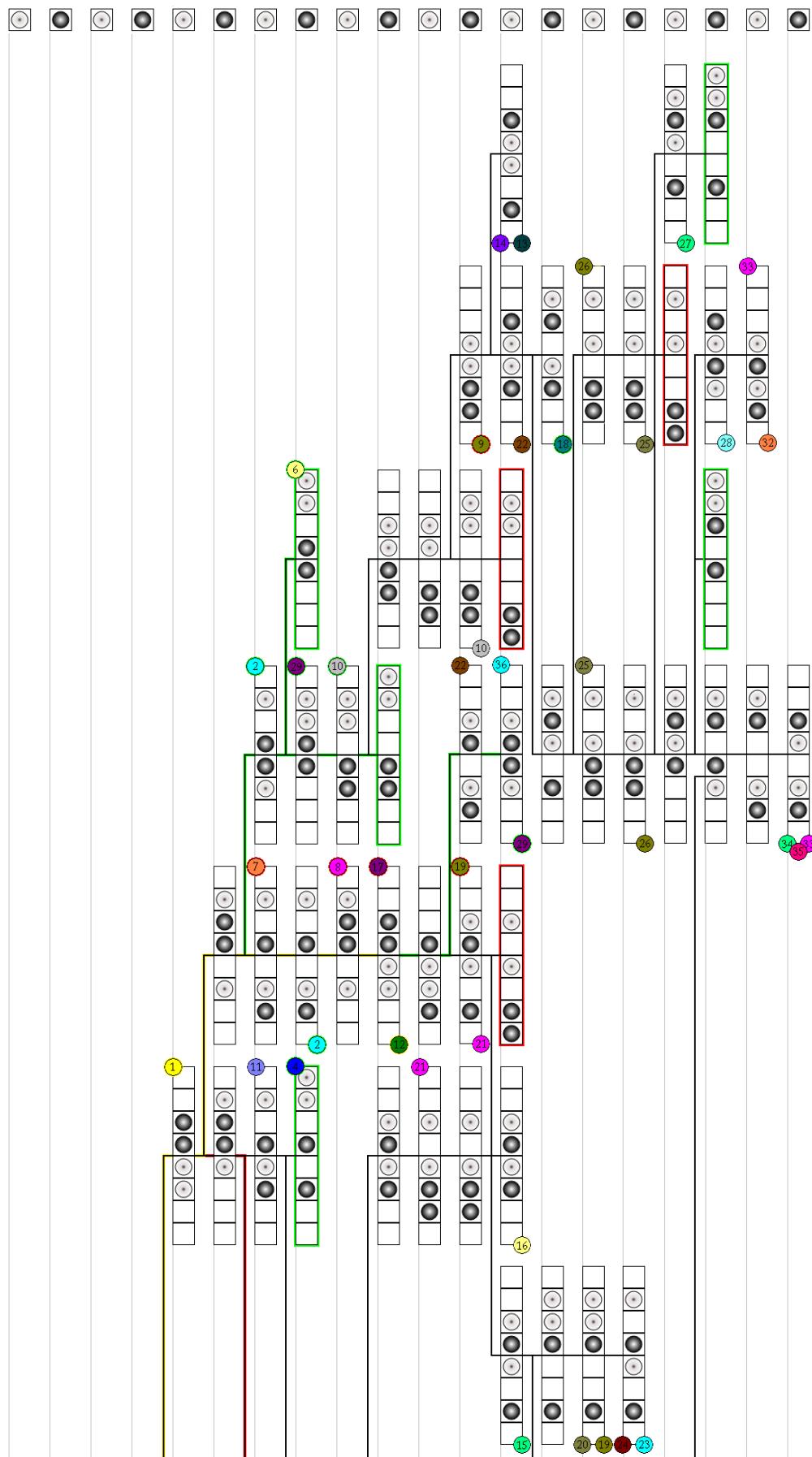
Dodatkowo program „Weles” pozwala na rozgrywkę człowiek kontra człowiek, człowiek kontra konkretny algorytm albo człowiek kontra wcześniej wygenerowany plik strategii.

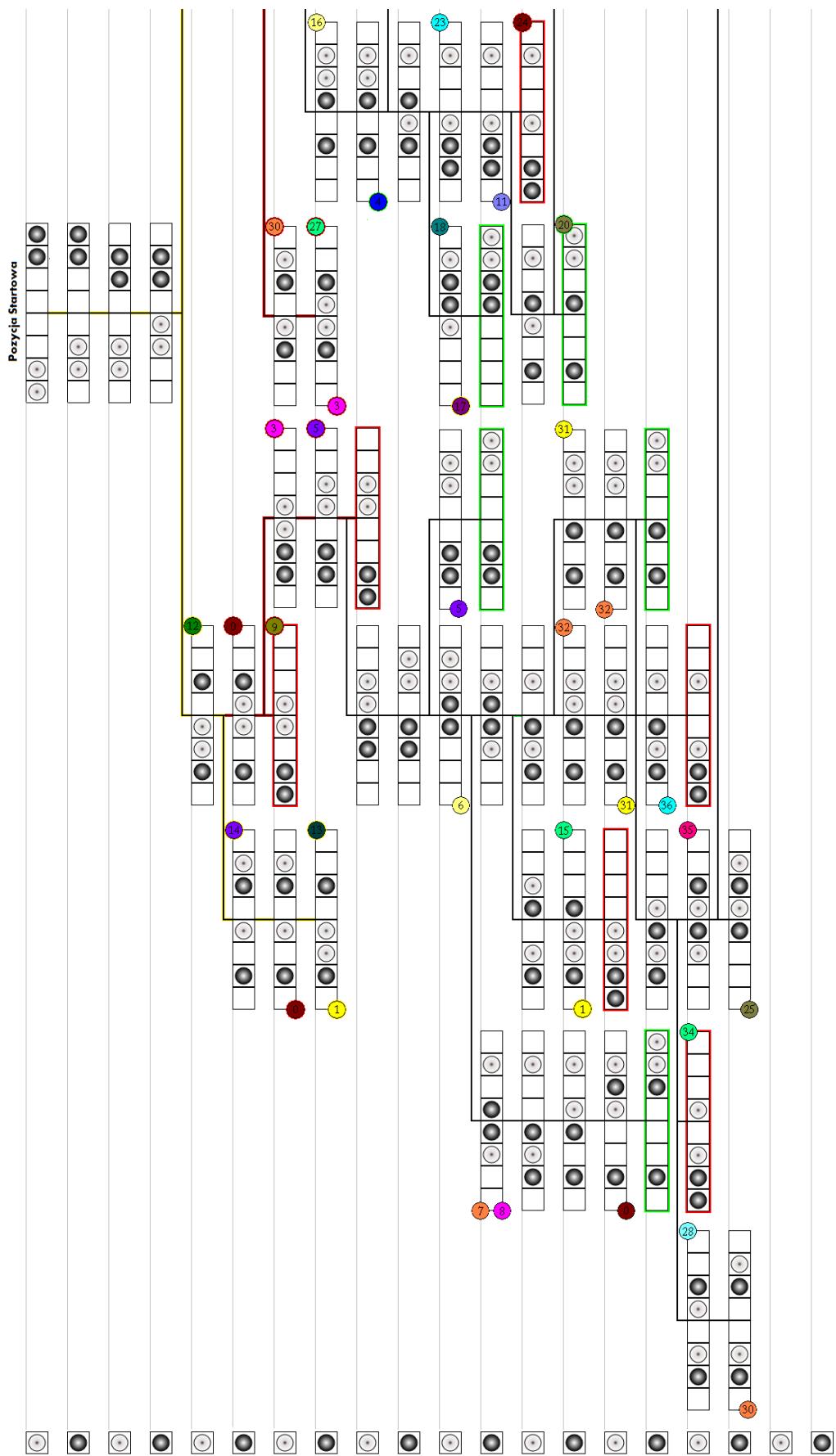
Dodatek A

Przestrzeń stanów gry *Skoczki*

Na rys. A.1 i A.2 zamieszczono wszystkie możliwe stany gry *Skoczki* wraz z wszystkimi możliwymi przejściami. Dotyczy on wariantu B1 oraz B2, w których nie można się przesuwać ale można wykonywać wieloskoki. Możliwość przeskakiwania przez pozycje startowe, bądź jej brak, nie zmienia drzewa gry dla rozważanych rozmiarów planszy. Ponieważ w przedstawionym digrafie gry występuje wiele pętli, w celu zachowania czytelności zamieniono niektóre łuki na pary liczb. Wspomnianej zamianie uległy tylko łuki prowadzące do stanów, które pojawiły się już wcześniej w digrafie (posiadają krótszą ścieżkę prowadzącą od korzenia). Taka zamiana polega na dodaniu dwóch liczb. Liczba na dole dowolnego stanu informuje o tym, że wychodzi z niego pewien łuk. Analogicznie liczba u góry dowolnego stanu informuje, że do wierzchołka dochodzi pewien łuk.

Stany przedstawiające pozycje reprezentujące wygraną gracza rozpoczynającego, oznaczone są zieloną ramką. Analogicznie pozycje reprezentujące jego przegrana, oznaczone są ramką czerwoną. Łuki oznaczone żółtym kolorem oznaczają rozgrywkę prowadzoną według optymalnych strategii obydwu graczy. Jest to rozgrywka remisowa. Remis uzyskiwany jest poprzez nieskończone odwlekanie możliwości przegranej przez obydwu graczy. Pozostałe oznaczone łuki (w czerwonym i zielonym kolorze) wraz z incydentnymi do nich stanami gry pokazują minimalny zbiór, który trzeba odwiedzić aby ustalić wartość korzenia tej wersji gry. Kolory pionków ponad górną i poniżej dolnej części grafu oznaczają kolor aktywnego gracza.

RYSUNEK A.1. Pełna przestrzeń stanów gry *Skoczki*(1 × 8) (górną połową)



RYSUNEK A.2. Pełna przestrzeń stanów gry *Skoczki*(1×8) (dolna połowa)

Dodatek B

Zasady omawianych gier

B.1. Aukcja o dolara

- Gra wieloosobowa o sumie niezerowej.
- Gra jest formą aukcji, której przedmiotem jest banknot jednodolarowy.
- Aukcję wygrywa osoba, która zaoferuje największą kwotę.
- Każda z osób biorących udział w aukcji musi uiścić zadeklarowaną kwotę, nawet jeśli jej nie wygrała.

B.2. Awari

- Gra dwuosobowa z gatunku gier Mankala.
- Rozgrywka toczy się na planszy składającej się z 2 rzędów pól i dwóch magazynów, każdy po jednym z boków planszy, pokazanej na rys. 2.17 w roz. 2.3.
- Na początku rozgrywki na każdym polu znajduje się identyczna liczbą pionów, w tym przypadku cztery.
- Gracze zajmują miejsca po przeciwnych, dłuższych bokach planszy (północny i południowy).
- Każdy z graczy posiada 6 pól w przyległym do niego rzędzie oraz magazyn znajdujący się po prawej stronie.
- Ruch polega na wybraniu przez gracza własnego, niepustego pola i rozsianiu jego zawartości.
- Rozsianie polega na zdjęciu wszystkich pionów z pola i umieszczanie po jednym na kolejnych sąsiadujących polach z pominięciem magazynów oraz (jeśli rozsiewane jest ponad jedenaście pionów) pola źródłowego.
- Rozsiewanie odbywa się w kierunku przeciwnym do ruchu wskazówek zegara.

- Jeżeli ostatni pion w z rozsiania ląduje na polu przeciwnika zawierającym dwa lub trzy pionki (włącznie z zasianym) to podlegają one biciu.
- Jeżeli na polu sąsiadującym z polem, na którym nastąpiło bicie (w kierunku przeciwnym do rozsiewania) także znajdują się tylko dwa lub trzy piony i należą one do przeciwnika to te piony również są bite.
- Kontynuacja bicia następuje dopóki warunki opisane we wcześniejszym punkcie są spełnione.
- Bicie polega na zdjęciu zbitych z bitego pola i umieszczeniu ich w magazynie gracza.
- Jeżeli ostatni pion z rozsiania ląduje na polu należącym do gracza rozsiewającego, to gracz kończy ruch nie przechwytując żadnych pionków.
- Gra kończy się, jeśli po stronie jednego z graczy nie pozostał ani jeden pion, po czym jego przeciwnik umieszcza piony z wszystkich swoich pól w magazynie.
- Wygrywa gracz, który zgromadził większą liczbę pionów w swoim magazynie.

B.3. Czwórki

- Gra dwuosobowa z gatunku gier połączeniowych.
- Rozgrywka toczy się na pionowo usytuowanej planszy o 7 kolumnach i 6 wierszach, pokazanej na rys. 2.6 w roz. 2.3.
- W grze obowiązują prawa grawitacji.
- Gracze wykonują kolejno ruchy polegające na wrzuceniu piona swojego koloru do dowolnej kolumny zawierającej przynajmniej jedno wolne pole.
- Ten z graczy, który ułoży cztery pionki swojego koloru poziomo, pionowo bądź skośnie wygrywa.
- Jeśli żadnemu z graczy nie uda się ułożyć czwórki, gra kończy się remisem.

B.4. Dakon-6

- Gra dwuosobowa z gatunku gier Mankala.
- Rozgrywka toczy się na planszy składającej się z 2 rzędów pól i dwóch magazynów, każdy po jednym z boków planszy, pokazanej na rys. 2.17 w roz. 2.3.

- Na początku rozgrywki każde pole wypełnione jest identyczną liczbą pionów równą liczbie pól po stronie każdego z graczy, w tym przypadku sześcioma.
- Gracze zajmują miejsca po przeciwnych, dłuższych bokach planszy (północny i południowy).
- Każdy z graczy posiada 6 pól w przyległym do niego rzędzie oraz magazyn znajdujący się po prawej stronie.
- Po uzgodnieniu rozpoczynającego, rozpoczyna się pierwsza runda, polegająca na wybraniu przez gracza własnego, niepustego pola i rozsianiu jego wartości.
- Rozsianie polega na zdjęciu wszystkich pionów z pola i umieszczaniu po jednym na kolejnych sąsiadujących polach z pominięciem magazynów oraz (jeśli rozsiewane jest ponad jedenaście pionów) pola źródłowego.
- Rozsiewanie odbywa się w kierunku przeciwnym do ruchu wskazówek zegara.
- Jeżeli ostatni pion z rozsiania ląduje na nie pustym polu to wszystkie piony z tego pola są wyjmowane i rozsiewane dalej według dotyczasowych reguł.
- Jeżeli ostatni pion z rozsiania ląduje w magazynie gracza to gracz może wykonać kolejny ruch.
- Jeżeli ostatni pion z rozsiania ląduje na pustym polu po stronie gracza to ten pion oraz wszystkie piony z przeciwnego pola są zdejmowane i umieszczane w magazynie gracza, po czym gracz kończy swój ruch.
- Jeżeli ostatni pion z rozsiania ląduje na pustym polu po stronie przeciwnika to gracz kończy ruch nie przechwytując żadnych pionków.
- Runda kończy się, jeżeli jeden z graczy nie może wykonać poprawnego ruchu, po czym wszystkie pozostałe piony z pól graczy są przemieszczane do ich magazynów po czym zostają rozdane.
- Rozdanie polega na umieszczaniu, poczynając od lewego pola pionów z własnego magazynu w liczbie identycznej jak na początku gry (w tym przypadku jest to sześć).
- Jeżeli liczba pionów w magazynie jest zbyt mała aby zapełnić wszystkie pola te, które nie mogą zostać wypełnione zostają oznaczone i nie biorą udziału w kolejnej rundzie.

- Piony, na które nie wystarczyło miejsca na planszy lub nie można nimi zapełnić jeszcze jednego pola, pozostają w swoim magazynie.
- Gracz, który wygrał ostatnią rundę rozpoczyna kolejną.
- Gra kończy się jeśli pod koniec rundy jeden z graczy posiada niewystarczającą liczbę pionów aby wypełnić jedno własne pole.
- Wygrywa gracz, który zgromadził większą liczbę pionów w swoim magazynie.

B.5. Dylemat więźnia

- Gra dwuosobowa o sumie niezerowej.
- Dwie osoby (więźniowie) są podejrzane o popełnienie przestępstwa.
- Więźniowie nie mogą się z sobą kontaktować.
- Każdy z nich może przyznać się do winy bądź milczeć.
- Jeżeli obydwaj będą milczeć to dostaną minimalną karę (6 miesięcy więzienia).
- Jeżeli tylko jeden będzie milczeć, to dostanie on maksymalną karę (10 lat więzienia), a wspólnik wyjdzie na wolność.
- Jeżeli obydwaj będą zeznawać, to dostaną po połowie maksymalnej kary (5 lat więzienia).

B.6. Fanorona

- Gra dwuosobowa.
- Rozgrywka toczy się na planszy składającej się z czterdziestu 45 pól, pokazanej na rys. 2.20 w roz. 2.3.
- Każdy z graczy dysponuje 22 pionkami (białe bądź czarne).
- Grający białymi rozpoczyna grę.
- Gracze wykonują ruchy na zmianę.
- Ruch pionkiem polega na przesunięciu go na sąsiednie wolne pole połączone linią.
- Bicie może odbywać się przez podejście lub wycofanie.

- Bicie przez podejście ma miejsce, jeśli pionek gracza po wykonaniu ruchu sąsiaduje z pionkiem przeciwnika znajdującym się na polu połączonym linią równoległą do kierunku ruchu pionka. Biciu podlega wspomniany pionek przeciwnika i wszystkie kolejne jego pionki sąsiadujące bezpośrednio z bitym pionkiem w linii równoległej do kierunku podejścia.
- Bicie przez wycofanie ma miejsce, jeśli pionek gracza przed wykonaniem ruchu sąsiaduje z pionkiem przeciwnika znajdującym się na polu połączonym linią równoległą do kierunku ruchu pionka. Biciu podlega wspomniany pionek przeciwnika i wszystkie kolejne jego pionki sąsiadujące bezpośrednio z bitym pionkiem w linii równoległej do kierunku wycofania.
- Jeśli można wykonać kilka bić jednym ruchem (podchodząc do pewnego pionka możemy wycofywać się od innych), gracz decyduje się na jedno z nich.
- Jeśli nastąpiło bicie, gracz kontynuuje grę.
- Bicie jest obowiązkowe, wyjątek dotyczy pierwszego posunięcia każdego z graczy, wtedy można bić tylko raz.
- Jeśli gracz nie może wykonać żadnego ruchu to traci kolejkę.
- Ten z graczy, który nie ma już żadnej bierki na planszy przegrywa.

B.7. Go-Moku

- Gra dwuosobowa z gatunku gier połączeniowych.
- Rozgrywka toczy się na planszy składającej się z 361 pól, gdzie polem jest miejsce przecięcia się linii siatki składającej się z dziewiętnastu linii poziomych i pionowych. Rozgrywkę można również prowadzić na planszy 255 polowej pokazanej na rys. 2.7 w roz. 2.3.
- Grający czarnymi rozpoczyna grę kładąc swojego piona na środkowym polu (h8).
- Gracze wykonują kolejno ruchy polegające na postawieniu piona swojego koloru na dowolnym, dotychczas nie zajętym polu.
- Ten z graczy, który ułoży *piątkę* czyli pięć i tylko pięć pionków swojego koloru w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie), wygrywa.

- Jeśli żadnemu z graczy nie uda się ułożyć *piątki*, a plansza jest zapełniona pionami, gra kończy się remisem.

B.8. Go-Moku wolne

- Obowiązują identyczne zasady jak w standardowym *Go-Moku* z tym wyjątkiem tego, że wygrywa ten z graczy, który ułoży pięć lub więcej pionków swojego koloru w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie).

B.9. Gra w życie

- Gra bezosobowa.
- Toczy się na nieskończonej planszy składającej się z kwadratowych pól (komórek), przylegających do siebie bokami.
- Każda komórka posiada ośmiu sąsiadów.
- Każda komórka może być żywa bądź martwa.
- Stan wszystkich komórek na planszy zmienia się co pewną ustaloną jednostkę czasu.
- Kolejny stan planszy jest zależny tylko od bezpośrednio poprzedzającego go stanu.
- Jeżeli w pewnym stanie martwa komórka posiada dokładnie trzech żywych sąsiadów to zostaje ożywiona w następnym stanie gry.
- Jeżeli żywa komórka ma mniej niż dwóch żywych sąsiadów to umiera z samotności.
- Jeżeli żywa komórka ma więcej niż trzech żywych sąsiadów to umiera z zatłoczenia.

B.10. Halma

- Rozgrywka toczy się na kwadratowej planszy o wymiarach 16 kolumn × 16 rzędów.
- Każdy z graczy dysponuje 19 pionkami (w przypadku gry dwuosobowej) albo 13 (w przypadku gry czteroosobowej).

- Planszę do gry wraz z początkową pozycją w wersji dla dwóch graczy pokazano na rys. 3.1 w sekcji. 3.1.1. W przypadku rozgrywki prowadzonej przez czterech graczy ustawiają oni swoje piony na polach zakreślonych czerwonymi liniami.
- Rozpoczynającego grę wybiera się przez losowanie.
- Gracze wykonują posunięcia na zmianę.
- Ruch pionka odbywa się w pionie, poziomie bądź po skosie.
- Istnieją dwa typy ruchów w grze: przesunięcie i przeskok.
- Przesunięcie pionka polega na zajęciu przez niego dowolnego, sąsiadniego, pustego pola.
- Przeskok pionka polega na przeskoczeniu dowolnego innego pionka, bądź nieprzerwanego bloku pionków. Mogą to być zarówno pionki własne jak i przeciwnika, lub dowolna ich kombinacja.
- Przeskok kończy się na pierwszym wolnym polu za przeskakiwanymi pionkami.
- Ruch jest obowiązkowy, o ile gracz ma możliwość go wykonać, w przeciwnym razie traci kolejkę.
- Jeśli po wykonaniu przeskoku pionkiem istnieje możliwość kolejnego przeskoku w dowolnym kierunku, to można (ale nie trzeba) taki ruch wykonać.
- Wygrywa ten z graczy, któremu jako pierwszemu uda się zająć swoimi pionkami wszystkie startowe pozycje przeciwnika.

B.11. Hex

- Gra dwuosobowa z gatunku gier połączeniowych.
- Rozgrywka toczy się na heksagonalnej planszy o wymiarach (11 x 11), pokazanej na rys. 2.1 w roz. 2.3.
- Gracze wykonują kolejno ruchy polegające na postawieniu piona swojego koloru na dowolnym, dotychczas nie zajętym polu.
- Ten z graczy, który połączy ciągłą linią dwa brzegi planszy własnego koloru wygrywa.

B.12. Hex konkurencyjny

- Obowiązują wszystkie zasady identyczne jak w normalnym Hexie.
- Wprowadzona zostaje zasada „jednego ruchu wyrównującego” polegająca na tym, że jeden z graczy robi ruch otwierający, po czym drugi gracz decyduje, który kolor należy do niego.

B.13. Kalah(6,4)

- Gra dwuosobowa z gatunku gier Mankala.
- Rozgrywka toczy się na planszy składającej się z dwóch rzędów pól i dwóch magazynów, każdy po jednym z boków planszy, pokazanej na rys. 2.17 w roz. 2.3.
- Na początku rozgrywki każde pole wypełnione jest identyczną liczbą pionów, w tym przypadku czterema.
- Gracze zajmują miejsca po przeciwnych, dłuższych bokach planszy (północny i południowy).
- Każdy z graczy posiada 6 pól w przyległym do niego rzędzie oraz magazyn znajdujący się po prawej stronie.
- Rozgrywkę rozpoczyna gracz południowy.
- Ruch polega na wybraniu przez gracza niepustego, własnego pola i rozsianiu jego zawartości.
- Rozsianie polega na wyjęciu wszystkich pionów z pola i umieszczanie po jednym w kolejnych sąsiadujących polach i jeśli wystąpi taka konieczność we własnym magazynie oraz polu źródłowym po pełnym okrążeniu.
- Rozsiewanie odbywa się w kierunku przeciwnym do ruchu wskazówek zegara.
- Jeżeli ostatni pion z rozsiania ląduje w magazynie gracza to gracz może wykonać kolejny ruch.
- Jeżeli ostatni pion z rozsiania ląduje na pustym polu po stronie gracza to ten pion oraz wszystkie piony z przeciwnego pola są zdejmowane i umieszczane w magazynie gracza, po czym gracz kończy swój ruch.

- Jeżeli ostatni pion z rozsiania ląduje na pustym polu po stronie przeciwnika to gracz kończy ruch nie przechwytując żadnych pionków.
- Gra kończy się, jeśli po stronie jednego z graczy nie pozostał ani jeden pion wtedy przeciwnik umieszcza piony z wszystkich swoich pól w magazynie.
- Wygrywa gracz, który zgromadził większą liczbę pionów w swoim magazynie.

B.14. Latrunculi

- Gra dwuosobowa.
- Rozgrywka toczy się na prostokątnej planszy o wymiarach (7 x 8).
- Każdy z graczy dysponuje 24 pionkami.
- Rozstawienie początkowe pionków na planszy pokazano na rys. 4.1 w sekcji 4.1.
- Pionki poruszają się i biją w pionie i poziomie, nigdy po skosie.
- Ruch pionkiem polega na przemieszczeniu go o dowolną liczbę wolnych pól.
- Bicie odbywa się przez pojmanie i polega na ustawienia dwóch własnych pionów po obydwu stronach nieprzerwanego rzędu bądź kolumny pionków przeciwnika. Następnie wszystkie pojmane pionki przeciwnika zostają zdjęte z planszy.
- Bicie następuje tylko w wyniku posunięcia gracza atakującego.
- Grający białymi rozpoczyna grę.
- Gracze wykonują ruchy na zmianę.
- Ten z graczy, który nie ma już żadnej bierki na planszy przegrywa, a jego przeciwnik wygrywa partię. W przypadku gry na planszy pozostały tylko dwie bierki należące po jednej do każdego z graczy, gra kończy się remisem.

B.15. Maharadża i Sepoje

- Gra dwuosobowa, wariant Szachów.
- Rozgrywka toczy się na tradycyjnej szachownicy o wymiarach (8 x 8).

- Gracz czarny gra tradycyjnymi bierkami szachowymi reprezentującymi Sepo-jów, natomiast gracz biały gra pojedynczą figurą – Maharadżą.
- Bierki gracza czarnego są rozstawione zgodnie z tradycyjnymi zasadami gry w szachy, natomiast grający Maharadżą ustawia go na dowolnym wolnym polu.
- Przykładową pozycję startową pokazano na rys. 2.28 w roz. 2.3.
- Bierki czarnego poruszają się zgodnie z tradycyjnymi zasadami gry w szachy za wyjątkiem promocji pionów i roszad, natomiast Maharadża może w każdym posunięciu wykonywać ruch Hetmana albo Skoczka.
- Rozpoczyna grający Maharadżą.
- Gracze wykonują ruchy na zmianę.
- Grający czarnymi wygrywa jeśli zablokuje lub zabije Maharadżę.
- Grający białymi wygrywa jeśli zamatuje czarnego Króla.

B.16. Młynek

- Gra dwuosobowa.
- Rozgrywka toczy się na planszy przedstawionej na rys. 2.8(a).
- Każdy z graczy dysponuje 9 pionkami (czarne bądź białe).
- Partia dzieli się na trzy etapy.
- W pierwszym etapie, który zawsze wynosi 18 posunięć, gracze na przemian dostawiają po jednym pionku na nie zajęte pola planszy.
- W drugim etapie każdy z graczy przesuwa jeden własny pionek na sąsiednie nie zajęte pole planszy.
- Trzeci etap rozpoczyna się, gdy jednemu z graczy pozostały jedynie trzy pionki, może on w swoim ruchu przestawiać dowolny własny pionek na dowolne nie zajęte pole gry.
- We wszystkich etapach gry możliwe jest bicie pionków przeciwnika.
- Bicie następuje, jeśli gracz ustawi trzy pionki własnego koloru w prostej linii, pionowo bądź poziomo, takie ustawienie nazywamy młynkiem. Może on wtedy zdjąć dowolny pionek przeciwnika.

- Jeśli gracz utworzy dwa młynki w jednym posunięciu, zdejmuje dwa piony przeciwnika.
- Gracz, któremu pozostały 2 piony na planszy lub który nie może wykonać ruchu przegrywa.

B.17. Nim

- Gra dwuosobowa.
- Na początku gry znajdują się trzy stosy.
- W skład pierwszego stosu wchodzą trzy, drugiego cztery oraz piątego pięć pionków.
- Ruch gracza polega na zdjęciu dowolnie wybranej liczby pionów z dowolnie wybranego stosu.
- Grę wygrywa gracz, który bierze ostatni pionek.
- Istnieje wiele wariantów gry.
- Zależnie od wariantu zmienia się liczba stosów oraz wchodzących w ich skład pionków, liczba możliwych do zdjęcia pionów przez gracza podczas wykonywania posunięcia oraz reguła czy gracz biorący ostatni pionek przegrywa bądź wygrywa grę.

B.18. Orzeł i reszka

- Gra dwuosobowa.
- Każdy z graczy, w tajemnicy przed rywalem, wybiera orła bądź reszkę.
- Określa się, który z graczy wygrywa jeśli dokonali identycznego wyboru, a kto jeśli dokonali wyboru przeciwnego.
- Gracze ujawniają swój wybór.

B.19. Pentomino

- Gra dwuosobowa, z gatunku gier puzzle.
- Rozgrywka toczy się na planszy o wymiarach (8 x 8).
- W skład gry wchodzi dwanaście pionów o różnych kształtach, każdy składający się z pięciu kwadratów, pokazane na rys. 2.9 w roz. 2.3.
- Gracze wykonują kolejno ruchy polegające na umieszczaniu dowolnego pionka tak, aby nie pokrywał wcześniej położonego i aby każdy z jego kwadratów leżał dokładnie na jednym kwadracie planszy.
- Gracz, który nie może wykonać ruchu przegrywa.

B.20. Quartostandardowe

- Gra dwuosobowa, z gatunku gier połączeniowych.
- Rozgrywka toczy się na planszy o wymiarach (4 x 4).
- W skład gry wchodzi szesnaście pionów, które są wszystkimi możliwymi kombinacjami wysokości (niski-wysoki), odcienia (ciemny-jasny), wypełnienia (wypełniony-dziurawy), kształtu (okrągły-kwadratowy) – przykładową planszę do gry pokazano na rys. 2.11 w roz. 2.3.
- Gracze wykonują kolejno ruchy polegające na umieszczaniu dowolnego pionka na dowolnym, dotychczas nie zajętym polu planszy.
- Ten z graczy, który pierwszy ułoży cztery pionki z tą samą cechą poziomo, pionowo bądź skośnie wygrywa.

B.21. Quarto *twist*

- Obowiązują identyczne zasady jak w standardowym Quarto z wyjątkiem tego, że pionka, którym gracz ma wykonać ruch wybiera przeciwnik.

B.22. Qubic

- Gra dwuosobowa, z gatunku gier połączeniowych.
- Rozgrywka toczy się na trójwymiarowej planszy o wymiarach (4 x 4 x 4) pokazanej na rys. 2.5 w roz. 2.3.

- Gracze wykonują kolejno ruchy polegające na umieszczaniu pionków swojego koloru na dowolnym, dotychczas nie zajętym polu kostki.
- Ten z graczy, który ułoży cztery pionki swojego koloru poziomo, pionowo bądź skośnie (możliwe jest 76 kombinacji) wygrywa.
- W przeciwnieństwie do podobnej gry *Score-four* tutaj nie obowiązują prawa grawitacji.

B.23. Renju

- Gra dwuosobowa z gatunku gier połączeniowych.
- Rozgrywka toczy się na planszy składającej się z 255 pól, gdzie pole to przecięcie się linii siatki, składającej się z piętnastu linii poziomych i pionowych pokazanej na rys. 2.16 w roz. 2.3.
- Rozpoczyna gracz, grający tymczasowo czarnymi, kładąc swojego piona na środkowym polu (h8).
- W drugim posunięciu grający, tymczasowo białymi, musi położyć swój pionek na jednym z ośmiu pól, sąsiadujących z polem środkowym.
- W trzecim posunięciu grający, tymczasowo czarnymi, musi położyć piona swojego koloru na dowolnym, dotychczas nie zajętym polu ale w centralnym kwadracie planszy o wymiarach 5x5 pól.
- Po trzecim posunięciu grający, tymczasowo białymi, musi się zdecydować na kolor pionów, którymi ma zamiar grać do końca partii.
- Czwarte posunięcie rozpoczyna gracz, który teraz gra białymi, polegające na postawieniu piona swojego koloru na dowolnym, dotychczas nie zajętym polu.
- W piątym posunięciu czarny proponuje białemu swoje dwa asymetryczne posunięcia, z których biały wybiera jedno.
- Od szóstego posunięcia gracze wykonują kolejno ruchy polegające na postawieniu piona swojego koloru na dowolnym, dotychczas nie zajętym polu.
- Ten z graczy, który ułoży *piątkę* czyli pięć pionków swojego koloru w nieprzewidzianym rzędzie (poziomo, pionowo bądź skośnie) wygrywa.
- Dodatkowo gracz grający czarnymi nie może wykonać następujących posunięć:

- Ustawić sześciu lub większej liczby kamieni swojego koloru w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie).
 - wykonać ruchu, który jednocześnie utworzy więcej niż jedną *czwórkę*. Gdzie *czwórka* to cztery piony jednego koloru w rzędzie (poziomo, pionowo bądź skośnie), które po jednym posunięciu mogą stać się *piątką*;
 - wykonać ruchu, który jednocześnie utworzy więcej niż jedną *trójkę*. Gdzie *trójka* to trzy piony jednego koloru w rzędzie, które po jednym posunięciu mogą stać się *silną czwórką*. Gdzie *silna czwórka* to cztery piony jednego koloru w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie) z wolnymi obydwoema końcami.
- Grający białymi wygrywa również, jeśli ułoży więcej niż pięć pionków swojego koloru w nieprzerwanym rzędzie (poziomo, pionowo bądź skośnie).
 - Jeśli żadnemu z graczy nie uda się ułożyć *piątki*, czarny nie wykonał posunięcia zakazanego i plansza jest zapełniona pionami, gra kończy się remisem.

B.24. Teeko

- Gra dwuosobowa, z gatunku gier połączeniowych.
- Rozgrywka toczy się na planszy składającej się z 25 pól, gdzie pole to przecięcie się linii siatki, składającej się z pięciu linii poziomych i pionowych, pokazanej na rys. 2.12 w roz. 2.3.
- Każdy z graczy dysponuje 4 pionkami (czarne bądź czerwone).
- Grający czarnymi rozpoczyna.
- Rozgrywa składa się z dwóch części.
- W pierwszej części gracze wykonują kolejno ruchy polegające na umieszczaniu pionków swojego koloru na dowolnym, dotychczas nie zajętym polu planszy.
- W drugiej części gracze wykonują kolejno ruchy polegające na przesunięciu swojego pionka na dowolne sąsiadujące wolne miejsce.
- Ten z graczy, który niezależnie od części rozgrywki, ułoży cztery pionki swojego koloru poziomo, pionowo bądź skośnie lub ustawi cztery pionki w narożnikach kwadratu o boku 2x2, 3x3, 4x4 lub 5x5 pól, wygrywa.

B.25. Tygrysy i Kozy

- Gra dwuosobowa.
- Rozgrywka toczy się na planszy składającej się z 25 pól, gdzie pole to przecięcie się linii siatki, składającej się z pięciu linii poziomych i pionowych.
- Planszę i pozycję początkową przedstawiono na rys. 2.22 w roz. 2.3.
- Grający tygrysami posiada 4 figury, na początku rozgrywki rozstawione w czterech narożnikach planszy.
- Grający kozami posiada 20 figur, na początku rozgrywki znajdujących się poza planszą.
- Ruchy mogą wykonywać zarówno tygrysy i kozy, bić mogą tylko tygrysy.
- Ruch jakąkolwiek figurą polega na przesunięciu jej na sąsiednie wolne pole połączone linią.
- Bicie tygrysem może wystąpić jeśli na sąsiednim polu, po skosie znajduje się koza przeciwnika, a kolejne pole w linii prostej jest wolne.
- W jednym ruchu tygrys nie może dokonać więcej niż jednego bicia.
- Podczas bicia tygrys przemieszczany jest na wolne pole za kozą przeciwnika, która zostaje zdjęta z planszy.
- Rozpoczyna grający kozami stawiając figurę na dowolnym wolnym polu.
- W pierwszej części rozgrywki, trwającej 39 posunięć, grający kozami może jedynie stawiać nowe figury na planszy, nie może poruszać już ustalonimi.
- Gracze wykonują ruchy na zmianę.
- Grający tygrysami wygrywa jeśli zabije 5 kóz.
- Grający kozami wygrywa, jeśli zablokują gracza tygrysami, nie będzie on mógł wykonać żadnego posunięcia w swoim posunięciu.

B.26. Warcaby angielskie

- Gra dwuosobowa, wariant Warcabów w USA znany pod nazwą *Checkers*.
- Rozgrywka toczy się na warcabnicy o wymiarach (8 x 8).
- Każdy z graczy dysponuje 12 pionkami.
- Rozstawienie początkowe pionków na planszy pokazano na rys. 2.18(a) w roz. 2.3.
- W grze występują dwa rodzaje bierek: pionki i damki.
- Na początku rozgrywki na planszy znajdują się tylko pionki.
- Zarówno pionki jak i damki poruszają się i biją po skosie, w grze wykorzystane są jedynie 32 ciemne (aktywne) pola z wszystkich 64.
- Cztery dostępne pola znajdujące się w ostatnim przeciwnym rzędzie dla każdego z graczy, to pola awansu.
- Każdy pionek z chwilą dotarcia do pola awansu staje się damką.
- Ruch pionkiem polega na przemieszczeniu go na sąsiednie wolne pole po skosie w kierunku pola awansu.
- Bicie pionkiem może wystąpić jeśli na sąsiednim polu, po skosie, znajduje się bierka przeciwnika, a kolejne pole w linii prostej jest wolne.
- Podczas bicia pionek przemieszczany jest na wolne pole za bierką przeciwnika, która zostaje zdjęta z planszy.
- Bicie pionkiem może odbywać się tylko w kierunku pola awansu.
- Ruch damką polega na przemieszczeniu jej na dowolne sąsiednie wolne pole po skosie.
- Bicie damką może wystąpić jeśli na sąsiednim polu, po skosie, znajduje się bierka przeciwnika, a kolejne pole w linii prostej jest wolne.
- Podczas bicia damka przemieszczana jest na wolne pole za bierką przeciwnika, która zostaje zdjęta z planszy.
- Jeśli istnieje możliwość bicia na początku ruchu, to jest ono obowiązkowe.

- Jeśli po jednym biciu ta sama bierka może wykonać kolejne bicie w dowolnym kierunku, to gracz musi, dopóki ten warunek jest spełniony, kontynuować serię bić.
- Jeśli istnieje kilka możliwości bicia to gracz może wybrać którykolwiek z nich (nie musi być najdłuższe), jednak musi tak dugo kontynuować sekwencję bić pionkiem, jak dugo jest to możliwe.
- Wyjątkiem od powyższej reguły jest osiągnięcie przez piona pola awansu, co definitywnie kończy ruch.
- Po każdym biciu, zbita bierka jest zdejmowana z planszy (nie można podczas jednej serii bić przeskoczyć wielokrotnie jednej bierki)
- Grający białymi rozpoczyna grę.
- Gracze wykonują ruchy na zmianę.
- Ten z graczy, który nie ma już żadnej bierki na planszy lub nie może wykonać żadnego ruchu przegrywa, a jego przeciwnik wygrywa partię. W innym przypadku gra kończy się remisem.

B.27. Wilk i owce

- Gra dwuosobowa o niesymetrycznych zasadach.
- Rozgrywka toczy się na warcabnicy o wymiarach (8 x 8).
- Pierwszy z graczy (grający wilkiem) dysponuje jednym białym pionkiem.
- Drugi z graczy (grający owcami) dysponuje czterema czarnymi pionkami.
- Zarówno wilki jak i owce poruszają się po skosie, w grze wykorzystane są jedynie 32 ciemne (aktywne) pola z wszystkich 64.
- Wilk może poruszać się o jedno pole (po skosie) w górę i w dół, owce poruszają się (po skosie) tylko do góry.
- W początkowym ustawieniu grający owcami umieszcza swoje piony w dolnym rzędzie warcabnicy, natomiast grający wilkiem umieszcza go na jednym z czterech pól górnego górnego rzędu.
- Grający wilkiem rozpoczyna grę.

- Gracze wykonują ruchy na zmianę.
- Grający wilkiem wygrywa jeżeli zajmie jedno z pól startowych owiec.
- Grający owcami wygrywa jeżeli doprowadzi do sytuacji w której wilk nie będzie mógł wykonać żadnego posunięcia.

Dodatek C

Zawartość płyty DVD-ROM

Na dołączonej płycie DVD zamieszczono:

- cyfrową wersję pracy magisterskiej przystosowaną do wydruku jednostronnego (plik – *mgr-one-sided.pdf*);
- cyfrową wersję pracy magisterskiej przystosowaną do wydruku dwustronnego (plik – *mgr-two-sided.pdf*);
- kod źródłowy pracy magisterskiej w języku L^AT_EX wraz z załącznikami;
- aplikację „Weles” (plik wykonywalny – *WelesGUI.exe* i biblioteka – *Weles.dll*);
- kod źródłowy aplikacji „Weles” (projekt w IDE Microsoft Visual Studio 2008 w języku C#);
- prezentację pracy magisterskiej w wersji pełnej (plik – *mgr-prezentacja.pdf*);
- prezentację pracy magisterskiej w wersji skróconej (plik – *mgr-prezentacja-trimed.pdf*);
- kod źródłowy prezentacji pracy magisterskiej w języku L^AT_EX wraz z załącznikami;
- darmowe środowisko .NET Framework 3.5 Service Pack 1 (pełny pakiet).

Na płycie jest umieszczony podpis opiekuna naukowego stwierdzający zgodność pracy w wersji elektronicznej z wersją papierową.

Bibliografia

- [1] J. D. Allen, *Connect FourTM*, URL <http://fabpedigree.com/james/connect4.htm>.
- [2] L. V. Allis, *A knowledge-based approach to connect-four. The game is solved: White wins*, Praca magisterska, Vrije Universiteit (1988).
- [3] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence*, Rozprawa doktorska, University of Limburg (1994).
- [4] L. V. Allis, H. J. van den Herik, M. P. H. Huntjens, *Go-Moku and Threat-Space Search*, Rap. tech. CS 93-02, Department of Computer Science, Faculty of General Sciences, Rijksuniversiteit Limburg, Maastricht, The Netherlands (1993).
- [5] L. V. Allis, H. J. van den Herik, M. P. H. Huntjens, *Go-Moku Solved by New Search Techniques*, w: *Computational Intelligence*, 12 str. 7–23 (1996).
- [6] V. V. Anshelevich, *The Game of Hex: An Automatic Theorem Proving Approach to Game Programming*, w: *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, str. 189–194, AAAI Press, Menlo Park, CA (2000).
- [7] E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways for your Mathematical Plays*, tom 2, A. K. Peters (1982).
- [8] E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways for Your Mathematical Plays*, tom 3, A. K. Peters (2003).
- [9] Y. Bjornsson, N. Burch, R. Lake, J. Culberson, P. Lu, J. Schaeffer, S. Sutphen, *Chinook - World Man-Machine Checkers Champion*, URL <http://www.cs.ualberta.ca/~chinook/>.
- [10] H. L. Bodlaender, *The Maharaja and the Sepoys*, URL <http://www.chessvariants.org/unequal.dir/maharaja.html>.
- [11] D. M. Breuker, *Memory versus Search in Games*, Rozprawa doktorska, Maastricht University (1998).

- [12] D. M. Breuker, H. J. van den Herik, J. W. H. M. Uiterwijk, L. V. Allis, *A solution to the GHI problem for best-first search*, Rap. tech. CS 97-02, Computer Science Department, Universiteit Maastricht, Maastricht, The Netherlands (1997).
- [13] M. Campbell, A. J. Hoane Jr., F. H. Hsu, *Deep Blue*, w: *Artificial Intelligence*, 134(1-2) str. 57–83 (2002).
- [14] R. Dawkins, *The Selfish Gene: 30th Anniversary Edition—with a new Introduction by the Author*, Oxford University Press, USA (2006).
- [15] S. T. Dekker, H. J. van den Herik, I. S. Herschberg, *Perfect knowledge revisited*, w: *Artificial Intelligence*, 43(1) str. 111–123 (1990).
- [16] J. Donkers, A. de Voogt, J. W. H. M. Uiterwijk, *Human versus Machine Problem Solving: Winning Openings in Dakon*, w: *Board Games Studies*, 3 str. 79–88 (2000).
- [17] B. Enderton, *Answers to infrequently asked questions about the game of Hex*, URL <http://www.cs.cmu.edu/~hde/hex/hexfaq>.
- [18] D. B. Fogel, *Evolving a checkers player without relying on human experience*, w: *Intelligence*, 11(2) str. 20–27 (2000).
- [19] R. Gasser, *Solving Nine Men’s Morris*, w: *Computational Intelligence*, 12 str. 24–41 (1996).
- [20] M. Grimminick, URL <http://www.xs4all.nl/~mdgsoft/draughts/stats/index.html>.
- [21] R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, J. van Rijswijck, *Solving 7×7 Hex: Virtual Connections and Game-State Reduction*, w: *Advances in Computer Games*, tom 263, str. 261–278, Kluwer Academic Publishers (2003).
- [22] P. Henderson, B. Arneson, R. B. Hayward, *Solving 8×8 Hex*, w: *International Joint Conference on Artificial Intelligence* (2009).
- [23] H. J. van den Herik, J. W. H. M. Uiterwijk, J. van Rijswijck, *Games solved: now and in the future*, w: *Artificial Intelligence*, 134(1-2) str. 277–311 (2002).
- [24] A. Holshouser, H. Reiter, *Mathematics and Informatics Quarterly*, w: *Quarto without the Twist*, 16(2) str. 50–55 (2005).

- [25] G. I. Jeroen, G. Irving, J. Donkers, J. W. H. M. Uiterwijk, *Solving Kalah*, w: *ICGA*, 23(3) str. 139–147 (2000).
- [26] A. Kishimoto, *Correct and efficient search algorithms in the presence of repetitions*, Rozprawa doktorska, University of Alberta (2005).
- [27] A. Kishimoto, M. Müller, *Df-Pn In Go: An Application To The One-eye Problem*, w: *Advances In Computer Games Many Games, Many Challenges*, str. 125–141, Kluwer Academic Publishers (2003).
- [28] A. Kishimoto, M. Mueller, *A Solution to the GHI Problem for Depth-First Proof-Number*, w: *7th Joint Conference on Information Sciences*, str. 489–492 (2003).
- [29] A. Kishimoto, M. Müller, *A general solution to the graph history interaction problem*, w: *AAAI'04: Proceedings of the 19th national conference on Artificial intelligence*, str. 644–649, AAAI Press / The MIT Press (2004).
- [30] K. Kryukov, *3x3 Chess*, URL <http://kirr.homeunix.org/chess/3x3-chess>.
- [31] K. Kryukov, *Endgame Tablebases Online*, URL <http://kirill-kryukov.com/chess/tablebases-online>.
- [32] R. Lake, J. Schaeffer, P. Lu, *Solving Large Retrograde Analysis Problems Using a Network of Workstations*, Rap. tech. TR-93-13, University of Alberta, Edmonton, Canada (1993).
- [33] D. N. L. Levy, D. F. Beal (red.), *Heuristic programming in artificial intelligence: the second computer olympiad*, Ellis Horwood (1991).
- [34] Y. J. Lim, J. Nievergelt, *Computing Tigers and Goats.*, w: *ICGA*, 27(3) str. 131–141 (2004).
- [35] A. Nagai, *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*, Rozprawa doktorska, Department of Information Science, University of Tokyo (2002).
- [36] W. C. Oon, Y. J. Lim, *An investigation on piece differential information in co-evolution on games using kalah*, w: *Proceedings of the 2003 Congress on Evolutionary Computation* (pod redakcją R. Sarker, R. Reynolds, H. Abbass,

- K. C. Tan, B. McKay, D. Essam, T. Gedeon), str. 1632–1638, IEEE Press, Canberra (2003).
- [37] H. K. Orman, *The Game of Pentominoes: A First Player Win*, w: *Games of No Chance*, tom 29, str. 339–344, Cambridge University Press (1994).
- [38] G. Owen, *Teoria gier*, PWN, Warszawa (1975).
- [39] O. Patashnik, *Qubic: $4 \times 4 \times 4$ tic-tac-toe*, w: *Mathematics Magazine*, 53 str. 202—216 (1980).
- [40] L. Pijanowski, *Przewodnik Gier*, ORENDA & DK Zakład Poligraficzno-Wydawniczy, Warszawa (1997).
- [41] E. Plonka, *Wykłady z teorii gier*, Wydawnictwo Politechniki Śląskiej, Gliwice (2001).
- [42] J. van Rijswijck, *Queenbee's home page*, URL <http://www.cs.ualberta.ca/~queenbee>.
- [43] J. van Rijswijck, *Search and Evaluation in Hex*, Rap. tech. T6G 2H1, University of Alberta, Edmonton, Canada (2002).
- [44] J. M. Robson, *Combinatorial Games with Exponential Space Complete Decision Problems*, w: *Proceedings of the Mathematical Foundations of Computer Science 1984*, str. 498–506, Springer-Verlag, London, UK (1984).
- [45] J. W. Romein, H. E. Bal, *Solving Awari with Parallel Retrograde Analysis*, w: *IEEE Computer*, 36(10) str. 26–33 (2003).
- [46] L. Russ, *The Complete Mancala Games Book: How To Play the World's Oldest Board Games*, Marlowe & Company, New York (2000).
- [47] J. T. Saito, G. Chaslot, J. W. H. M. Uiterwijk, H. J. van den Herik, *Monte-Carlo Proof-Number Search for Computer Go*, w: *Computers and Games*, tom 4630, str. 50–61 (2007).
- [48] M. Sakuta, H. Iida, *The Performance Of Pn^* , Pds , And Pn Search On 6×6 Othello And Tsume-Shogi*, w: *Advances in Computer Games*, 9 str. 203–222 (2001).

- [49] M. P. D. Schadd, M. H. M. Winands, J. W. H. M. Uiterwijk, H. J. van den Herik, M. H. J. Bergsma, *Best Play In Fanorona Leads To Draw*, w: *New Mathematics and Natural Computation*, 4(03) str. 369–387 (2008).
- [50] J. Schaeffer, *Checkers Is Solved*, w: *Science*, 317(5844) str. 1518–1522 (2007).
- [51] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen, *Solving Checkers*, w: *International Joint Conference on Artificial Intelligence (IJCAI)*, tom 19, str. 292–297 (2005).
- [52] J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, S. Sutphen, *Building the Checkers 10-piece Endgame Databases*, w: *ACG*, tom 263, str. 193–210 (2003).
- [53] J. Schaeffer, R. Lake, *Solving the Game of Checkers*, w: *Games of No Chance*, tom 29, str. 119–136, Cambridge University Press (1996).
- [54] M. Schijf, L. V. Allis, J. W. H. M. Uiterwijk, *Proof-Number Search and Transpositions*, w: *ICCA*, 17(2) str. 63–74 (2004).
- [55] U. Schwalbe, P. Walker, *Zermelo and the Early History of Game Theory*, w: *Games and Economic Behavior*, 34(1) str. 123–137 (2001).
- [56] M. Seo, H. Iida, J. W. H. M. Uiterwijk, *The PN*-search algorithm: application to tsume-shogi*, w: *Artificial Intelligence*, 129(1-2) str. 253–277 (2001).
- [57] C. E. Shannon, *Programming a computer for playing chess*, w: *Philosophical Magazine*, 41 str. 256–275 (1950).
- [58] L. Smolin, *The Life of the Cosmos*, Oxford University Press, USA (1999).
- [59] E. Trice, G. Dodgen, *The 7-piece Perfect Play Lookup Database for the Game of Checkers*, w: *Advances in Computer Games*, tom 263, str. 211–230, Kluwer (2003).
- [60] J. Tromp, *John's Connect Four Playground*, URL <http://homepages.cwi.nl/~tromp/c4/c4.html>.
- [61] E. Weisstein, URL <http://mathworld.wolfram.com/Teecko.html>.
- [62] E. van der Werf, H. J. van den Herik, J. W. H. M. Uiterwijk, *Solving Go on Small Boards*, w: *ICGA*, 26(2) str. 92–107 (2003).
- [63] J. Wágner, I. Virág, I.: *Solving Renju*, w: *ICGA*, 24 str. 30–34.

- [64] M. H. M. Winands, J. W. H. M. Uiterwijk, H. J. van den Herik, *An effective two-level proof-number search algorithm*, w: *Theoretical Computer Science*, 303(3) str. 511–525 (2004).
- [65] M. H. M. Winands, J. W. H. M. Uiterwijk, H. J. V. D. Herik, *PDS-PN: A New Proof-Number Search Algorithm*, w: *Computers and games*, tom 2883, str. 61–74 (2002).
- [66] L. Wittgenstein, G. E. M. Anscombe, E. Anscombe, *Philosophical Investigations: The German Text, with a Revised English Translation 50th Anniversary Commemorative Edition*, Wiley-Blackwell (1991).
- [67] J. Yang, S. Liao, M. Pawlak, *Another Solution for Hex 7 × 7*, Rap. tech., University of Manitoba, Winnipeg, Canada (2002).
- [68] J. Yang, S. X. Liao, M. Pawlak, *New Winning and Losing Positions for 7×7 HEX*, w: *Computers and games*, tom 2883, str. 230–248 (2002).

Skorowidz

- algorytm
 pn^2 -search, 82
 BTA Proof-Number Search, 95
 Df-pn, 110
 Minimax, 57
 Monte-Carlo Proof-umber Search,
 110
 NegaMax, 61
 PDS, 109
 PN*, 109
 Proof-Number Search, 71
- algorytmy
 best-first search, 70
 depth-first search, 57
- analiza wsteczna, 53
- argument z kradzeniem strategii, 16
- digraf
 acykliczny AND/OR, 67
 cykliczny AND/OR, 68
- dowód
 na brak remisu w grze Hex, 13
 na istnienie najbardziej dowodzącego
 wierzchołka, 74
- drzewo
 minimaksowe, 59
 negamaxowe, 61
 z odcinaniem α - β , 63
 AND/OR, 65
- funkcja indeksująca, 52
- gra
 Aukcja o dolara, 5, 155
- Awari*, 29, 155
 Bagh-Chal, 33, 169
 Czwórki, 18, 21, 156
 Dakon, 24, 156
 Dylemat więźnia, 158
 Fanorona, 32, 158
 Go-Moku, 20, 159
 Go, 40
 Gra w życie, 5, 160
 Halma, 46, 160
 Hex, 13, 39, 161
 Kalah, 27, 162
 Latrunculi, 163
 Młynek, 21, 47, 164
 Maharadża i Sepoje, 38, 163
 Nim, 5, 165
 Ohvalhu, 24
 Orzeł i reszka, 5, 165
 Othello, 36
 Pentomino, 22, 166
 Quarto, 23, 166
 Qubic, 18, 166
 Renju, 27, 167
 Skoczki, 113
 innego wariantu, 116
 Szachy, 37
 Teeko, 23, 168
 Tygrysy i Kozy, 33, 169
 Warcaby
 angielskie, 30, 48, 170
 polskie, 37
 Wilk i owce, 171

- bezstronna, 7
- deterministyczna, 8
- jednoetapowa, 8
- losowa, 8
- o sumie nie zerowej, 6
- o sumie zerowej, 6
- stronnicza, 7
- symetryczna, 7
- z kompletną informacją, 7
- z pełną informacją, 8
- heurystyka pustych ruchów, 20
- hipotezy
 - z 1990 roku, 19
 - z 2000 roku, 36
- interfejs
 - IAI, 119
 - IGameEngine, 119
 - IGameGUI, 119
 - ISolvingGUI, 119
- klasa
 - BlackPawn, 121
 - Board, 121
 - BoardState, 126
 - EndPosition, 121
 - GameManagement, 126
 - GameWindow, 121
 - JumpersEngine, 127
 - MovesGenerator, 130
 - Square, 121
 - StartPosition, 121
 - WhitePawn, 121
- liczba
 - dowodząca, 72
 - graczy, 5
 - obalająca, 72
- lista rozwiązań gier, 35
- podprzestrzenie stanów gry, 46
- popularność gry, 10
- posunięcie przekształcające, 85
- powtórzenie, 89
- problem grafu historii interakcji, 90
- przycinanie $\alpha-\beta$, 62
- rodzaj gry, 5
- rodzina gier Mankala, 24, 27
- rozwiązanie
 - lekkie gry, 12
 - silne gry, 12
 - ultra-lekkie gry, 12
- symetria
 - gry, 50
 - zasad gry, 7
- transpozycje
 - w DAG, 85
 - w DCG, 89
- trudność gry, 10
- twierdzenie Zermelo, 11
- wierzchołek
 - wyliczenie
 - natychmiastowe, 70
 - opóźnione, 70
 - bazowy, 95
 - bieżący, 83
 - bliźniaczy, 95
 - graniczny
 - niewyliczony, 71
 - wyliczony, 71
- najbardziej
 - dowodzący, 74
 - rozszerzenie, 70

- rozwinięcie, 70
 - terminalny, 71
 - transpozycyjny, 85
 - typu
 - AND, 66
 - OR, 66
 - wewnętrzny, 71
- złożoność
- drzewa gry, 9
 - obliczeniowa gry, 9
 - przestrzeni stanów gry, 9
- zbiór
- dowodzący, 72
 - obalający, 72
- zbieżność gry, 10